TURING 图灵程序设计丛书

MANNING

Ext JS in Action

Second Edition

Ext JS 实战

（第2版）



Jesus Garcia
[美] Grgur Grisogono 著
Jacob K. Andresen

卢雄飞 田雷鹏 郜忠富 译

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

本书由Ext JS社区三位专家合力打造，全面介绍了灵活而强大的Ext JS框架。全书不仅深入浅出地介绍了基础知识、工作原理和类系统，更结合完整的示例、丰富的插图解读了其核心组件，让你仿佛在专家的亲手指导下学习并实践。

通过学习，你将真正掌握这个成熟的JavaScript Web应用框架，用它丰富而漂亮的UI组件缩短开发周期，构建优雅布局，轻松管理单调乏味的样板文件，最大程度减少手工编码和浏览器兼容问题。更重要的是，你将学会快速创建和扩展桌面风格的Web应用，从而提升用户体验，开拓更为广阔的Ext JS应用前景。

主要内容：

- Ext JS的丰富特性，包括UI部件、数据存储、模型、代理等；
- 用Ext JS构建专业Web应用；
- 用模板消灭DOM碎片；
- 定制并构建Ext部件；
- 出色的UI设计。

本书适合任何想要学习并使用Ext JS且具有一定的JavaScript经验的开发人员阅读参考。

“学习Ext JS必备。”

——Loiane Groner，花旗银行

“一本内容丰富的专家指南，它将带你体验从新手到专家的成就感！”

——Jeet Marwah，gen–E

“全面掌控Ext JS 4.0，打造桌面风格的Web UI。”

——Efran Cobisi，微软MVP、MCSD

“从Ext JS试用到高效使用之间的重要一环。”

——Raul Cota，Virtual Materials集团

MANNING

图灵社区：iTuring.cn

热线：(010)51095186转600

分类建议 计算机/程序设计/Ext JS

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-39592-4

ISBN 978-7-115-39592-4

定价：79.00元

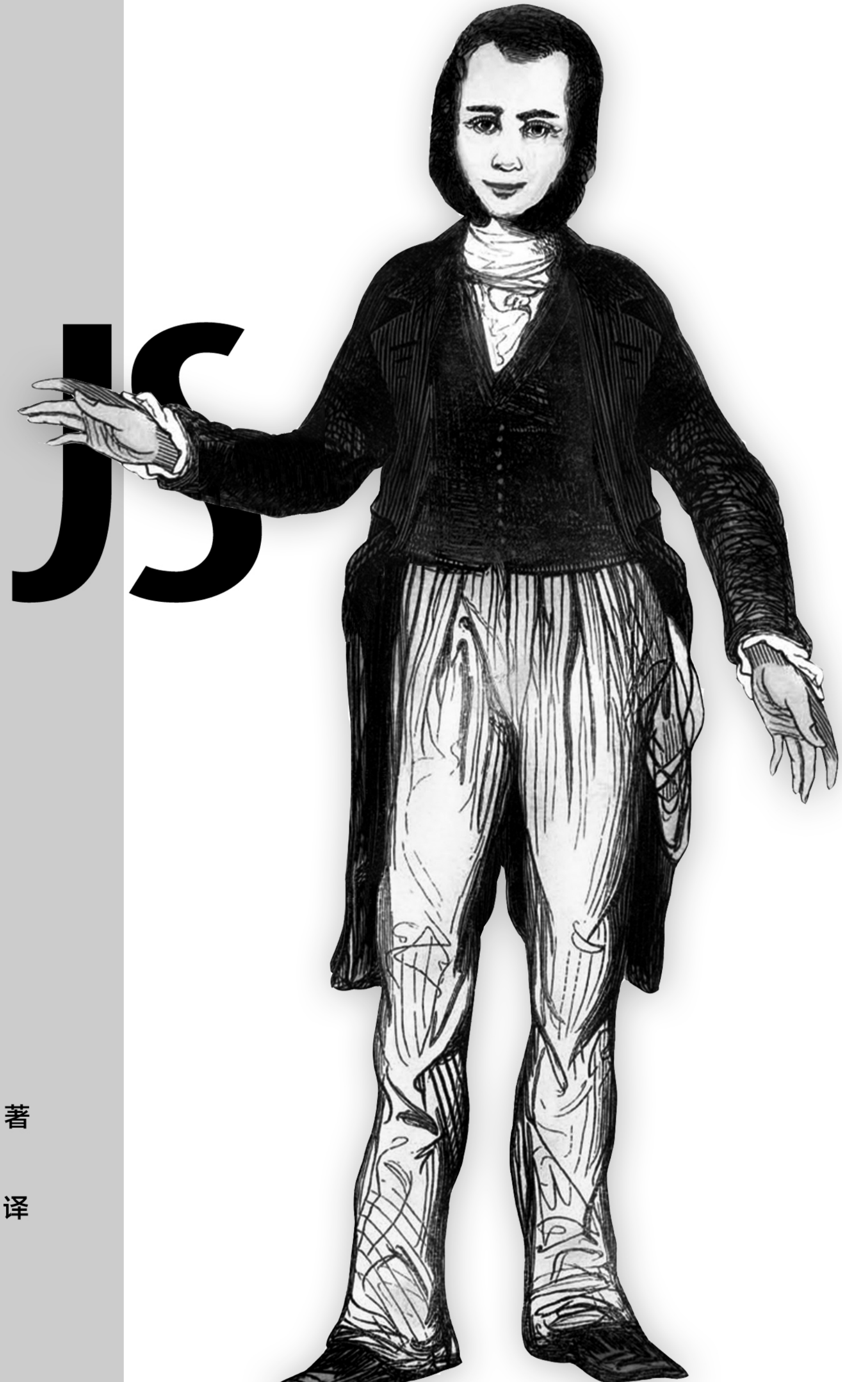
图灵程序设计丛书

Ext JS in Action

Second Edition

Ext JS 实战

（第2版）



Jesus Garcia

[美] Grgur Grisogono 著

Jacob K. Andresen

卢雄飞 田雷鹏 郜忠富 译

人民邮电出版社

北京

图书在版编目（CIP）数据

Ext JS实战 /（美）加西亚（Garcia,J.），（美）格丽索戈诺（Grisogono,G.），（美）安德烈森（Andresen,J.K.）著；卢雄飞，田雷鹏，郜忠富译. --2版. -- 北京 : 人民邮电出版社，2015.8
（图灵程序设计丛书）
ISBN 978-7-115-39592-4

Ⅰ. ①E… Ⅱ. ①加… ②格… ③安… ④卢… ⑤田… ⑥郜… Ⅲ. ①JAVA语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2015)第131404号

内 容 提 要

本书分三部分，基于 Ext JS 4.0 全面介绍如何开发具有原生风格的富桌面 Web 应用，辅以大量示例帮读者理解其组件和容器。第一部分是基础知识，讲解 Ext JS 的丰富特性（包括 UI 部件以及数据存储、模型和代理等支持类）、DOM 操作、组件和容器。第二部分全面介绍 Ext JS 部件的工作原理，内容涵盖 Ext JS 组件、布局管理器、表单面板、数据存储、网格面板、树形面板、图形图表、直接远程调用和拖放功能。第三部分介绍 Ext JS 类系统，并基于本书知识用 Sencha CMD 和 Ext JS MVC 系统开发应用，不仅能让你学会 Ext JS 框架的更高级功能（如定制的扩展、插件，以及类加载器），还能让你掌握构建和管理 Web 应用的坚实理论。

本书适合各水平阶段的 Web 开发人员学习参考。

◆ 著　　[美] Jesus Garcia　Grgur Grisogono
　　　　Jacob K. Andresen
　译　　卢雄飞　田雷鹏　郜忠富
　责任编辑　李松峰　毛倩倩
　执行编辑　柏华元　陈德伟
　责任印制　杨林杰
◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
　邮编　100164　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　印刷
◆ 开本：800×1000　1/16
　印张：22.5
　字数：532千字　　2015年 8 月第 1 版
　印数：1－2 500册　　2015年 8 月北京第 1 次印刷
著作权合同登记号　图字：01-2014-1135号

定价：79.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

版 权 声 明

前　言

我在职业生涯中开始接触Sencha要追溯到2006年。它就是如今称为Ext JS的Sencha桌面JavaScript框架的前身，在当时顶多算是一项实验。入门后没多久，我就开始对这个快速发展的框架所提出的设计模式着迷不已。不过，更重要的是，我爱上了这个茁壮成长而又乐于回报的开发人员社区。

受到这个社区里很多活跃成员的激励，我决定自己也做一份贡献，开始每周花几十个小时回答问题、发帖子，并最终发表了教学视频。那个时候真的是很有意思，社区中涌现出了大量设计模式。

本书这一版（初版于2010年面世）反映了Ext JS 4.0所引领的一个桌面Web前端开发的新纪元。这一版介绍了一套极其健壮的类系统，并提供了很多扩展自JavaScript的功能。此外，还有一套设计非常精良的事件系统、数据包、用户界面（User Interface，UI）和模型–视图–控制器架构（Model View Controller，MVC），而且Ext JS 4.0有一套强大的框架，可以让你开发出能使用很多年的应用。

我们很高兴与你分享Ext JS知识，希望你能乐在其中。

——Jesus Garcia

致　　谢

我们要向以下各位表示诚挚的谢意。

- Sencha开发者社区——没有这一社区，根本就不可能有这本书。
- Sebastian Sterling——这本书的出版耗时比我们预想的要长得多。作为我们在Manning出版社的主要策划编辑，他对我们的写作严格要求，并帮助我们呈现最好的内容，谢谢他的辛苦工作。我们还要感谢Frank Pohlman，他在本书出版的最后阶段给我很大的帮助，并最终将本书交付印制。
- Manning出版社的制作团队——他们真是太棒了！能有机会与他们合作我们深感荣幸，不管是出版这本书还是我们之前的书，我们都非常重视他们这么多年所做的工作。衷心感谢！
- MEAP（Manning出版社的早期试读项目）读者——谢谢他们在作者在线论坛（Author Online）上给出的指正和建议。
- 审校者——他们在本书出版过程中反复阅读书稿，并提出了宝贵的洞见和反馈来帮助我们完善本书。感谢Bradley Meck、Brian Crescimanno、Brian Daley、Brian Forester、Chad Davis、Darragh Duffy、Efran Cobisi、Jeet Marwah、John J. Ryan III、Loiane Groner、Mary Turner、Raul Cota、Robby O'Connor和Todd Hill。
- Doug Warren——事实证明，他在本书出版过程中对内容（包括代码）所做的技术审校和详尽检查对我们弥足珍贵。非常感谢！

Jesus Garcia

写这本书，我很是费了一番工夫，但如果没有大家的帮助和贡献我肯定无法做到。在此，谨对以下各位表示感谢。

- 我的妻子Erika。这本书酝酿了几年时间，当人们恭贺我的时候，往往不会意识到如果没有你给我无尽的支持，我是写不成这本书的。我全身心地深爱着你，并因生命中有你而感恩不已。
- 我的儿子Takeshi和Kenji。我不会忘记写作期间在屋中跑来跑去玩耍的你们，感谢你们容忍我长时间写作。你们是我如此努力工作的动力所在，我深爱着你们。
- Mitchell Simoens——我为有你这样的朋友而感恩。我为你在职业发展和个人发展两方面的成就感到骄傲。请永远记住用知识去挑战自己的极限。

- Abe Elias——看到你多年来在Sencha不断成长，直至带领一支顶尖的专业工程师团队，我很是惊叹。每当我谈到牛人的时候，总是会最先想到你。请继续牛下去！
- Grgur Grisogono——我的生活因你的出现变得更加美好。我珍视我们的友谊，并期待在未来一直与你同行。
- Jacob Andresen——你对这份书稿做出了重要的贡献，感谢你为撰写各章内容所付出的辛劳。
- Don Griffin——谢谢你允许我参与关于Sencha Cmd和其他Ext JS相关工具的讨论。

Grgur Grisogono

我要感谢我挚爱的妻子Andrea，以及我的孩子Laurenco和Paulina，感谢他们对我长期的支持和鼓励。他们给我极大的支持与关爱，让我可以有动力并集中精力写作本书，时刻精力充沛。我真诚地感谢那些了不起的审校人员，他们抽出宝贵的时间细致审校本书，帮助我为整个开发者社区优化本书内容。

我还要向Modus Create团队表达谢意，感谢他们支持我并让我面临新的挑战，促使我成为一名更好的专业人士。我要特别感谢Sencha和它的核心团队工程师们，他们帮助保证本书内容紧跟Ext JS和Sencha Cmd的更新步伐，并帮助保证了本书质量。

我要向两位最富盛名的Ext JS开发者社区成员、最了不起的人物致以最诚挚的谢意，他们就是本书的合著者Jesus Garcia和Jacob Andresen。他们是我完美的队友，不知疲倦地引领着本书的写作过程，审校着书中的各种谬误。

最后，感谢我永生难忘的好朋友、好榜样、好同事Jesus Garcia，感谢他给予我能量，感谢他的鼎力支持和耐心。他的深刻洞见始终令我折服，他观察入微，妙想连连。

Jacob Andresen

首先，我要感谢Jesus Garcia让我参与本书的写作。撰写本书不仅让我有机会学习写作技巧，而且得以了解本书中有关技术细节的始末。我还想感谢Grgur Grisogono为本书所付出的心血，以及他在国际Sencha开发者社区中所做的工作。

说到开发者社区，当然少不了来自斯堪的纳维亚的Mats Bryntse、Fredric Berling和Emil Pennlöv，感谢他们带给我的那些美好时光。

最重要的是，感谢我的妻子Anita，感谢她对我长久以来彻夜编程的理解与支持。

关于本书

本书旨在向读者介绍灵活而强大的桌面框架Ext JS。本书将详细介绍这一框架的基础知识，以及如何使用Sencha Cmd开发和部署成品应用。通读本书，你应该可以开发出健壮的桌面Web应用，并了解Ext JS 4.0的众多新特性。

目标读者

本书将向开发人员讲解如何利用Ext JS创建具有原生风格的富桌面Web应用。虽然Ext JS被主题化和高度定制，但本书面向那些在规范实施中主要从事编程工作的人员。

我们假定你对网站与Web服务器之间如何交互已有了初步的了解。想要最有效地编写健壮的响应式应用，你需要拥有扎实的技术背景，掌握诸如HTML、CSS、JavaScript和JSON之类的核心技术；我们只会在第13章详细介绍这些核心技术，并在那里探讨JavaScript的原型继承（它是Ext JS类系统的先决条件）。

软件要求

本书将带你查看很多实践范例。为更好地掌握这些内容，你需要安装以下软件。

- **Web服务器**　我们推荐Apache HTTPD或Microsoft IIS。
- **智能集成开发环境（IDE）**　我们推荐安装Webstorm或Aptana。
- **Sencha Cmd副本**　下载网址是www.sencha.com/products/senchacmd/download。

差不多就是这些了！

导读

本书旨在引导你学习Ext JS，且这一版根据Ext JS 4.0更新了很多内容。我们将集中介绍Ext JS提供的众多丰富特性，包括UI部件以及数据存储、模型和代理等支持类。本书共14章。

第1章简单介绍Ext JS框架。我们将全面审视这个框架，探讨很多常用的部件。

第2章旨在带你试用这一框架。我们要好好看看这个框架是如何交付给你的，并辨析它的内容。我们还要遍历DOM（文档对象模型）操作的基础知识，并且逐步提高难度，用Ext JS模板引擎Template和XTemplate来渲染DOM中的数据。

第3章讲的是`Component`和`Container`，两者都是Ext JS用户界面的基础类。我们将讨论组件的生命周期，并介绍如何用`Container`和它的工具方法来管理和查询子组件。

第4章以第3章的内容为基础讨论核心UI组件，比如面板、窗口、消息框和标签面板。这些都是扩展自`Container`的基本部件，让你的UI实现比`Container`所能提供的更多的功能。

第5章包含Ext JS提供的多种布局管理器，后者用于整合组件在屏幕上的显示。这一章将让你学会用众多Ext JS部件构建复杂的用户界面。

第6章围绕表单面板（form panel）和多种输入栏展开介绍。我们将探讨如何用输入栏设置验证，以及如何用表单面板加载和保存数据。

第7章集中讨论Ext JS数据包。你将了解核心数据类`Model`、`Proxy`、`Reader`和`Store`，以上这些类都是用来给各种UI组件提供数据的。

第8章在第7章的基础上讲解网格面板（grid panel）。我们将探讨支持网格面板的多个类，与此同时介绍如何使用很多常见的实现模式。

第9章全面探讨Ext JS树形面板（tree panel）。我们将深入了解如何使用数据`TreeStore`类支持树形面板部件的分级数据，并在这一章的最后学习通过Ext JS菜单来操作树数据。

第10章介绍了Ext JS的图形和图表包。在探索如何用Ext JS Draw API在画布上绘图的同时，你将能绘制简单的图形。之后，你将学会实现Ext JS提供的众多图表。

第11章集中讨论如何用Ext JS直接进行Web远程调用。我们将探索如何才能把服务器端逻辑与客户端集成，以容许服务器代码控制向客户端发出的API调用。

第12章介绍如何用Ext JS实现拖放。我们将介绍如何实现基本的拖放类，进而深入探讨如何用网格、树和数据视图来实现拖放功能。

第13章集中讨论Ext JS类系统。一开始我们要介绍基本的JavaScript原型继承，为你之后开发Ext JS类打下基础。你将了解如何扩展Ext JS组件并开发该框架的插件。

第14章将带你全程感受如何用Sencha CMD和Ext JS MVC系统开发应用。你将学习设置基本的应用脚手架，用MVC开发一个应用，然后生成测试和产品构建。

代码约定

本书中的所有源代码都使用等宽字体显示，以区别于其他文本。在很多代码清单中，我们都对代码作了注释以指出关键概念。我们尝试对代码进行排版，通过谨慎添加换行符和缩进使之能够契合书中现有的页面空间。不过有时候很长的代码行会包含续行符。

获取最新示例

我们特意将本书中的示例设计为易于浏览的形式。每一章都有独立的文件夹，每个示例都根据其所对应的代码清单命名。

我们将努力确保示例与框架更新同步。要获得示例的最新版本，你可以下载GitHub版本库：

https://github.com/ModusCreateOrg/extjs-in-action-examples。你还可以在Manning出版社网站www.manning.com/ExtJSinActionSecondEdition下载一个包含代码示例的压缩文件[①]。

作者在线

购买了本书英文版的读者均可以免费访问Manning出版社专门维护的一个网上论坛（Author Online），并可以发表评论、提出技术问题，从作者和其他论坛用户那里获得帮助。你可以通过网页www.manning.com/ExtJSinActionSecondEdition进入和订阅该论坛。完成注册后，你可以了解如何使用该论坛、该论坛所能提供的帮助，以及论坛上的行为规范。

Manning出版社承诺为读者和作者提供一个进行深入对话的场所，但不对作者的参与程度做要求，他们对于该论坛的贡献均是出于自愿且无报酬的。我们建议读者尽量向作者提一些具有挑战性的问题，让他们保持参与的兴趣！

① 本书中文版的读者还可注册iTuring.cn，至本书页面免费下载。——编者注

关于封面图

本书封面上的人物插图题为“Le voyageur”，意思是“旅行商”。这幅插图摘自Sylvain Maréchal编著并于19世纪在法国出版的区域着装习俗概要（共四卷）。这套概要中，每一张插图都是全手工精细绘制和着色。其中丰富多彩的插图生动地为我们展现了区区二百年前这个世界的城镇和地区在文化上与现在有多大不同。人们说着不同的方言和语言，彼此隔绝分立，在街头或者是乡间，人们的衣着就清晰表明了他们住在那里，做哪一行或者身份地位如何。

从那之后，人们的着装几经更迭，曾经鲜明的区域间的着装多样性逐渐消亡。现在仅看着装已经很难分辨住在不同大陆的居民，更别说是住在不同城镇或者地区的人了。或许，我们已经用文化上的多样性换取了更为多样的个人生活——当然也是为了更丰富、更快速的科技生活。

在这个任何计算机书几乎都是一个模子的年代，Manning用类似这样插图集中的作品把两个世纪前区域生活的丰富多样性呈现在封面上，借此颂扬计算机行业的创造力和首创精神。

目　　录

第二部分 Ext JS组件

第三部分 构建一个应用

Part 1 第一部分

Ext JS 4.0 入门

本书全面地阐释并展示了如何使用强大的Ext JS框架开发JavaScript应用。大量的实用示例将帮助你了解它的组件和容器，更重要的是了解如何综合使用它们。

第1章概述Ext JS 4.0的新特性。这一章也会讲解该框架的基本概念和部件，带你编写“Hello World”应用。第2章阐述开发任意Ext JS应用所需的基础要素，比如初始化、DOM元素操作，以及用Ajax服务器数据注入HTML片段模板。第3章介绍组件，并介绍UI构建块的生命周期，这里的UI构建块包括视口、面板、菜单、选项卡、数据网格、动态表单、风格化的弹出窗口，以及管理子项的容器和布局控件。

读完第一部分，你将全面了解Ext JS的工作原理，并做好探索Ext JS框架众多组成部件的准备。

第 1 章

功夫在框架外

1

本章内容

- 了解Ext JS 4.0的新内容
- 获取源代码
- 解析“Hello world”示例

来设想一种情况：你受命开发一个应用，其中要用到很多典型的UI（用户界面）部件，比如菜单、选项卡、数据网格、动态表单和风格化弹出式窗口。此时需要一种可以通过编程来控制部件位置的东西，这就意味着它必须有布局控件。可能你还想要详细且有组织的集中式文档，以便更轻松地学习和掌握这一框架。最后，应用必须外观成熟，能够尽快进入测试阶段，也就是说没有太多时间来摆弄HTML和CSS。在编写原型的第一行代码之前，你必须先确定开发应用前端的方法。你会选什么呢？

对市面上常见的流行框架和库做过一番调查之后，你很快就发现它们全都可以操控DOM，但其中只有两个拥有成熟的UI部件：雅虎用户界面（Yahoo! User Interface，YUI）和Ext JS。

初窥YUI，你也许感觉无需再找其他选择。尝试过它的示例，你会发现它的部件看起来很成熟，但是还达不到专业品质，也就是说还得修改CSS。那可不行！接下来再看它的文档。文档是集中式的，技术上也很准确，但用户友好度远远不够。你会发现它的所有滚屏都需要对应一个方法或者一个类。有些类甚至因为左侧导航窗格太小而被裁减。

本章我们要来好好看看Ext JS，了解一些构成这一框架的部件。在我们对Ext JS有一个整体的了解之后，可以下载它，并用它小试牛刀。

1.1 初识 Ext JS

为了开发一个拥有一套富UI控件的富网络应用（Rich Internet Application，RIA），你转用Ext JS，结果发现Ext JS独树一帜地提供了一套丰富的DOM工具和部件。关于示例的那一页或许会让你颇感兴奋，但幕后的内容才是最激动人心的。Ext JS带有一整套的布局管理工具，让你全面掌控，可以根据需求组织和操作用户界面。而在它下面一层有所谓的组件模型和容器模型，它们在管理UI的构建方式方面都起着非常重要的作用。

组件模型和容器模型

组件模型和容器模型在Ext JS管理UI方面起着重要作用，也正是Ext JS区别于其他Ajax库和框架的部分原因。组件模型决定了UI部件在所谓的**组件生命周期**中是如何被实例化、渲染和销毁的。容器模型控制着部件如何管理（或者**容纳**）其他子部件。这些是了解Ext JS框架的两个关键部分，也正是我们要在第3章花很多时间讨论这两个主题的原因。

在这个框架里，几乎所有的UI部件都是高度自定义的，可以选择启用和禁用某些特性、覆盖某些函数，以及使用某些自定义扩展和插件。conjoon就是一个充分利用Ext JS的Web应用示例。图1-1展示了conjoon实际使用中的一幅截图。

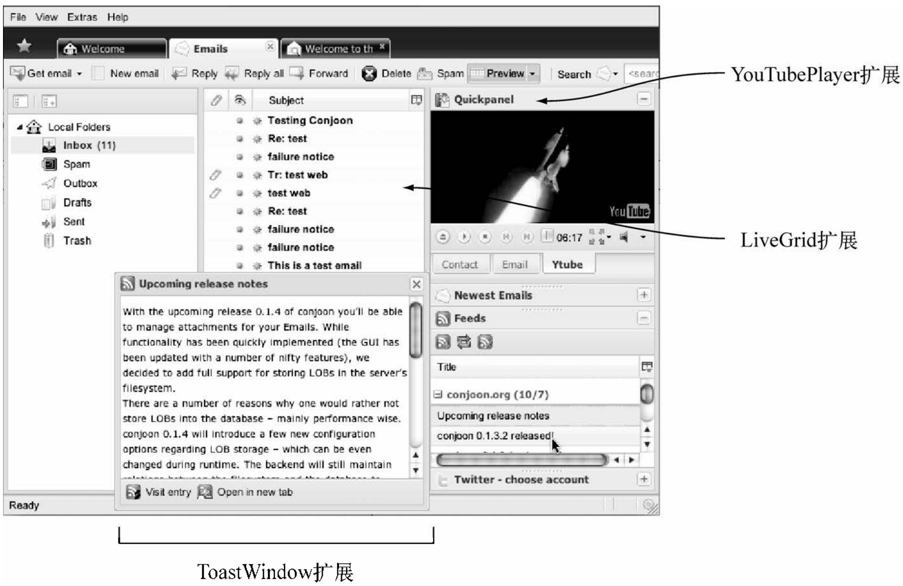


图1-1　conjoon是一个开源个人信息管理器，是使用Ext JS框架管理一个使用全部浏览器视口的UI的绝佳示例。下载地址：http://conjoon.org/

conjoon是一个开源个人信息管理器，可以被视为用Ext JS开发的Web应用的典型示例。它使用了几乎该框架的所有原生UI部件，并展现了该框架跟诸如YouTubePlayer、LiveGrid 和ToastWindow之类的自定义扩展集成得有多好。

现在你已经知道Ext JS可以用来创建一个全页Web应用。显而易见，我们可以利用这个框架做很多事情。你很快就会发现这个框架非常庞大，而且它的API文档很有用。

说到API文档，让我们换换口味，简单地了解一下它。

1.1.1　丰富的API文档

使用4.0版Ext JS框架的API文档是经过改进焕然一新的。初次打开API文档时，就可以感觉到

该框架的精致。和其他与之竞争的框架不同，Ext JS的API文档使用它自己的框架构建了一个简洁易用的文档工具，这个工具通过Ajax来生成API文档。

我们将解析API的所有特性，并讨论这个文档工具中用到的一些组件。图1-2展示了Ext JS的API文档应用中用到的一些组件。



图1-2　Ext JS API文档中包含大量信息，是深入学习组件和部件的一个好资源

这个API文档工具里充满了优质的图形用户界面（Graphical User Interface，GUI）元素，包含了6个最常用的部件，包括文本输入框（text input field）、树形面板、标签面板、主面板和工具栏（toolbar），外加嵌入式按钮。

> **浏览历史功能**
>
> Ext JS 4.0的文档现在加入了浏览历史功能。也就是说，你可以用浏览器的前进和后退按钮在API文档导航栏之间前后切换。

我知道你肯定在想这些是什么，有什么用。让我们花点儿时间讨论一下这些部件，然后再继续看后面的内容。

文本输入框这个部件封装了原生浏览器文本输入表单控件，增加了诸如校验之类的特性。在API文档中，它被用于对树形面板实施实时搜索，样式可自定义。我们将在第4章深入介绍标签面板。

树形面板部件以树状形式展示分级数据，类似于Windows资源管理器显示硬盘文件夹一样。标签面板提供了一种方法，可以在用户界面上显示多个文档或者组件，但每次只允许激活一个文档或者组件，虽然都在API文档中，但只显示一个项目。

主面板是Ext JS里的工作狂。它很灵活，涵盖很多可显示内容的区域，包括停靠栏（dock）和内容体（content body）。停靠栏就是通常放置诸如工具栏之类部件的地方，而内容体就是通常渲染内容或者子部件的区域。在API文档中，内容体包含了Ext JS框架的文档。

Toolbar类提供了一种展现按钮和菜单等常用UI组件的方法，这种情况下，它也能包含任何Ext.form.Field子类。可以把工具栏视为一个放置流行操作系统和桌面应用上常见的“文件–编辑–查看”菜单的地方。

API非常易用。要查看一个文档，只要点击树上的类节点即可。这将调用一个Ajax请求，取得所要的类的文档。每一个类文档都是一个HTML片段（而非一个完整的HTML页面）。

所以文档非常详实。但在快速应用开发方面呢？Ext JS能缩短开发周期吗？

1.1.2　用预制部件快速开发

Ext JS可以帮助你从概念快速跃升到原型，因为其中所需的很多UI元素它都已经预制好以备集成。拥有这些预制的UI部件，就不用再去设计建造相应的功能，因此可以节省大量时间。多数情况下，UI控件是高度可自定义的，可以根据应用的需求进行修改。

1.2　你需要知道的

虽然用Ext JS开发并不一定非得是Web应用开发方面的专家，但开发人员在尝试用这个框架编写代码以前，应该有一些这方面的核心竞争力。

这类核心技能最首要的，就是对于超文本标记语言（HTML）和层叠样式表（CSS）的基本理解。在这些技术方面有一定的经验非常重要，因为跟任何其他的JavaScript UI库一样，Ext JS也是用HTML和CSS来建造UI控件和部件的。它的部件看起来或许跟典型的现代操作系统控件很相像，但最终还是归结为浏览器里的HTML和CSS。

因为JavaScript是Ajax的粘合剂，所以需要学习者有坚实的JavaScript编程基础。同样，你不一定非得是专家，但应该充分掌握一些关键概念，比如说数组、引用和作用域。如果你熟悉对象、类和原型继承等面向对象JavaScript基础，那就更好了。如果刚接触JavaScript，那你走运了。JavaScript几乎自互联网诞生之初就已存在。W3Schools.com是一个初学JavaScript的好地方，那里提供大量的免费在线教程，甚至还有一些沙盒支持在线把玩JavaScript，你可以访问http://w3schools.com/JS/ 亲身体验。

如果受命开发服务器端的代码，那就需要一个Ext JS的服务器端解决方案进行交互以及存储数据。为保存数据，你要知道如何通过选择的服务器端语言与数据库或者文件系统交互。

当然了，可选的解决方案范围非常广。在这本书里，我们不会拘泥于某一种特定的语言，而是使用http://ExtJSinaction.com中的在线资源，在那里我们已经为你做好了服务器端的工作。这样一来，你只要专心学习Ext JS就好了。在此过程中，我们将提供特定的API URL地址供使用。

要探索Ext JS，我们首先带你纵览这个框架的全貌，了解它的功能类别。

1.3　Ext JS 部件之旅

Ext JS主代码库诞生于2010年初Sencha Touch的开发期间，后者是世界上第一个HTML5移动

框架（2010年11月发布）。Sencha Touch造就了Ext JS的支柱，也就是“Sencha平台”（参见图1-3），它包含很多Ext JS和Sencha Touch都使用的关键特性。这些共同特性包括DOM和事件管理、组件模型，以及布局，我们将在这本书的后面章节中详细介绍。

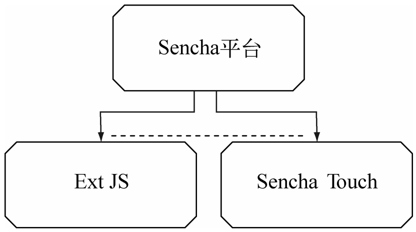


图1-3　Ext JS 4.0和Sencha Touch都衍生自Sencha平台，后者是Sencha系列HTML5框架的共同基础

Ext JS框架不但提供UI部件，而且提供了一系列其他特性。这些特性主要分布在7个主要的功能区：核心、UI组件、Web远程调用、数据服务、拖放、图形图表和通用工具集。图1-4中展现了这7个目标领域。

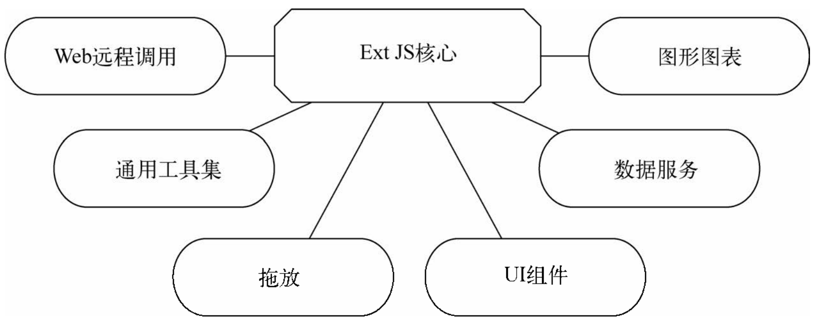


图1-4　Ext JS类的7个功能区

了解都有哪些不同的功能区以及它们能做什么，这可以为开发应用带来优势，所以我们要花点时间来讨论一下。

- 核心

第一个特性集是Ext JS 核心，它包括很多基本特性，比如Ajax通信、DOM操作，还有事件管理。所有其他部分都依赖于框架的核心，但核心并不依赖于任何其他部分。

- UI组件

UI组件包含所有与用户有交互的部件。

- Web远程调用

Web远程调用是用JavaScript远程执行在服务器上所定义和暴露的方法调用的一种手段，这些方法调用通常称为远程过程调用（Remote Procedure Call，RPC）。对于那些希望把服务器端方法暴露给客户端，而又不想为Ajax方法管理操心的开发环境来说，这很是便利。这个包称为Ext

Direct。

❑ 数据服务

数据服务部分处理所有的数据需求，包括数据抓取、解析以及把信息载入数据存储。利用Ext JS数据服务类，可以读取数组、可扩展标记语言（Extensible Markup Language，XML）和JSON。JSON是一种迅速成为客户端与服务器通信标准的数据格式。数据存储通常是给UI组件提供数据。

❑ 图形图表

这个全新的工具包包含了Ext JS跨浏览器绘图引擎，它与矢量标记语言（Vector Markup Language，VML）和可缩放矢量图形（Scalable Vector Graphic，SVG）兼容。利用图形包（Draw），你可以生成自己的数据可视化效果，但它的主要用途是作为图表包（Charting）的基础。图表包中包括了很多常用的图表，包括笛卡儿图表（条形图、折线图、柱状图等）、饼图、区域图、散点图和其他图表。

> **试用JSON！**
>
> 虽然JSON已经存在了很多年，但如果这是第一次听说它的话，我们建议你访问http://json.org，这是了解这种无所不在的数据交换格式的最佳信息源。如果你有兴趣学习在选择的服务器端语言中实施JSON，有无数JSON实施方法可选，其中大多数都有在线文档和讲解。我们建议用类似“PHP JSON”之类的关键词在谷歌上搜索了解一下。

❑ 拖放

拖放就像是Ext JS内部的一个迷你框架，你可以用它在页面上为Ext JS组件或者任何HTML元素赋予拖放功能。它里面包含了管理全部拖放操作的所有必要构件。拖放是一个复杂的话题，我们将用第13章和第14章两章来探讨这个主题。

❑ 工具集

工具集部分包括了那些可以帮你更轻松实施某些常规任务的绝佳工具类。其中的一个例子是`Ext.util.Format`，它让你可以轻松地格式化或者转换数据。另一个很不错的工具是CSS单例模式，它让你可以创建、更新、交换和移除样式表，还可以请求浏览器更新它的缓存规则。

现在你对Ext JS框架主要的功能区应该有了一个大体的认识，花点儿时间来了解一下Ext JS提供的一些常用UI部件吧。

1.3.1　容器和布局初探

虽然我们会在第3章详细探讨这两个主题，但还是应该先在这里花点儿时间大致了解一下。容器和布局这两个术语在本书中被大量使用，因此我们希望确保你在继续阅读之后的内容前，至少对它们有一个基础的认识。之后，我们将开始对UI库可视化组件的探索。

❑ 容器

容器是可以管理一个或者更多个子项的部件。一个子项通常来说，就是由一个容器或者父项

管理的任何部件或者组件，也就是父子范式。你已经在API中看到过这样的情况。标签面板是一个管理一个或多个子项的容器，我们可以通过标签面板访问其子项。请记住这个名词，因为当你开始深入学习使用框架的UI部分时会经常用到它。

- 布局

布局是由一个容器实现的，以直观地组织该容器内容体里的子项。Ext JS的库里有多达33个布局！好消息是只需要学习其中的13个，而我们将在第5章中对它们进行详细探讨，介绍每个布局的具体情况。

现在你对于容器和布局已经大致有了了解，让我们来看一些容器的作用。在图1-5中，你可以看到`Container`的两个子类——`Panel`和`Window`，它们分别与其他类建立了父子关系，这展现了`Container`类和多种布局的强大功能。

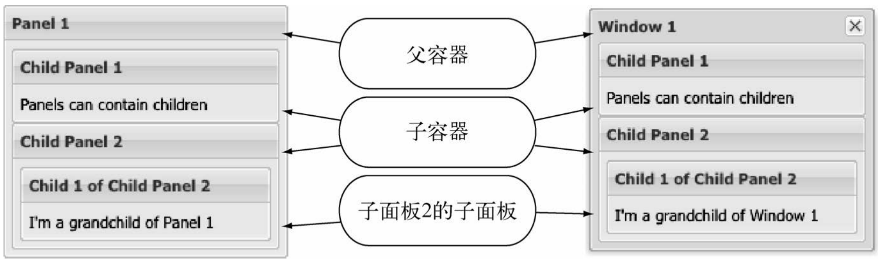


图1-5 在这里可以看到两个父级`Container`——`Panel`（左）和`Window`（右）——管理着一些子项，包括嵌套的子项

图1-5里的`Panel`（左）和`Window`（右）分别管理着两个子项。每个父容器的子面板1中都包含HTML。名为“子面板2”的子项都使用AutoLayout并各管理着一个子面板，AutoLayout是默认的容器布局。这种父子关系是Ext JS所有UI管理的关键要素，之后将在本书中一再涉及和强调。

你已经了解了容器如何管理子项，以及如何通过布局来直观地组织它们。在掌握了这些重要的概念之后，请来看一下其他容器的运作。

1.3.2 其他容器的运作

在学习`Container`的过程中，你已经看到了`Panel`和`Window`子类。图1-6展示了`Container`其他的一些常用子类。

在图1-6中，可以看到表单面板、标签面板、窗口、工具栏和输入框容器等部件。表单面板用`BasicForm`类封装输入框，用一个`form`元素封装其他子项。所有这些部件都被一个`Ext.window.Window`实例包含。

在第6章我们会花一些时间构建一个复杂的用户界面，让你更多地了解表单面板。接下来，我们来看看Ext JS框架所提供的数据呈现部件。

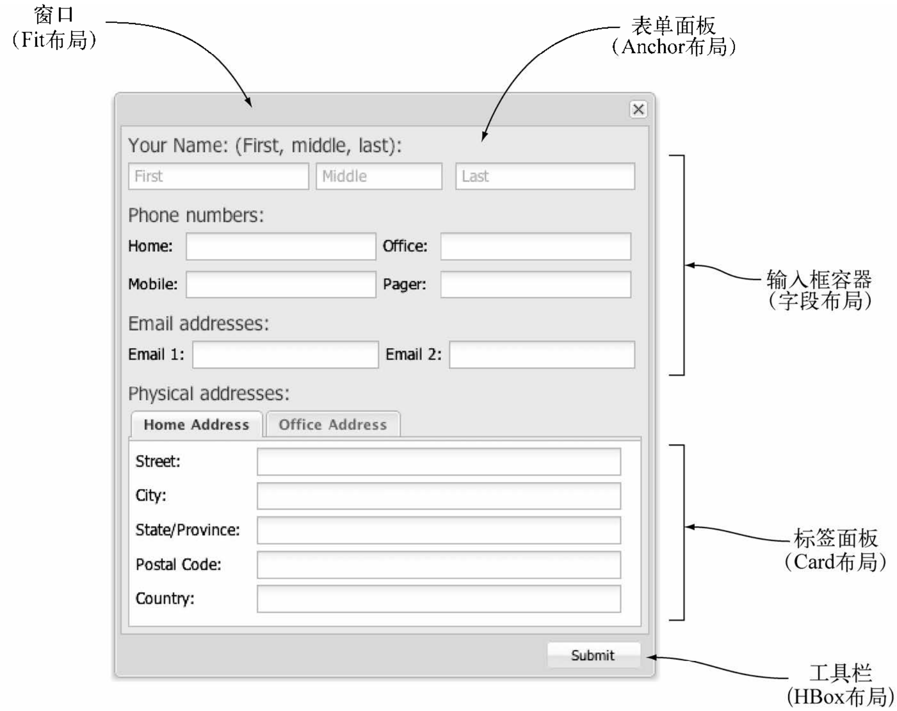


图1-6　用来构成这个UI窗口的常用Container子类——FormPanel、TabPane、FieldContainer和Toolbar——以及多种布局。我们将在第6章介绍表单的时候构建这个窗口

1.3.3　数据绑定视图

你已经了解到，Ext JS框架的数据服务部分是用来读取和解析数据的。Ext JS 4.0有很多绑定数据存储的部件，称为视图。你会用到很多视图，其中包括数据视图、网格面板和树形面板。如果应用程序需要用图表，那么告诉你一个好消息，Ext JS框架里所有图表都被视作视图并且绑定数据存储。图1-7是Ext JS的网格面板在实际使用中的一幅截图。

新近重构的GridPanel是Panel的一个子类，以类似表格的格式呈现数据，但它的功能经过扩展远超传统的表格，提供了可分类、可调尺寸和可移动列标题，以及像RowSelectionModel和CellSelectionModel这样的选择模型。你可以随心所欲地定制它的观感，并配合一个分页工具栏把大数据集分割并分页显示。它包含了很多特性和插件，让你可以进行逐行或逐格编辑，以及诸如锁定一列之类的操作。图1-8中所示的数据视图呈现了市面上多款不同手机的照片和其他相关信息。

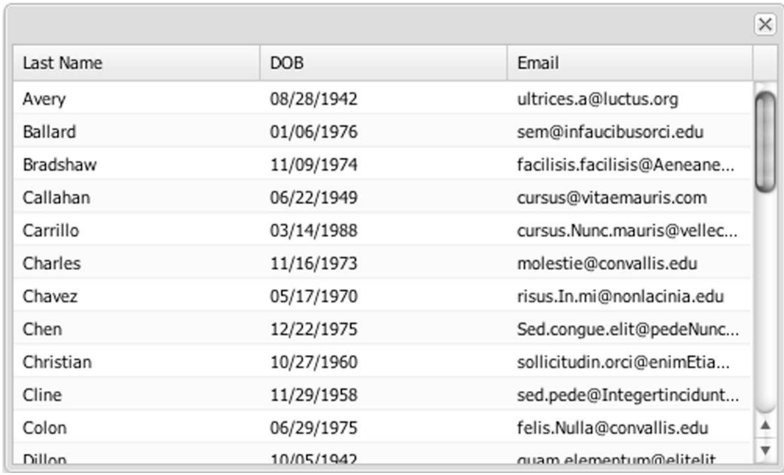


图1-7　Ext JS软件开发工具包（Software Development Kit，SDK）Buffered Grid示例中的网格面板

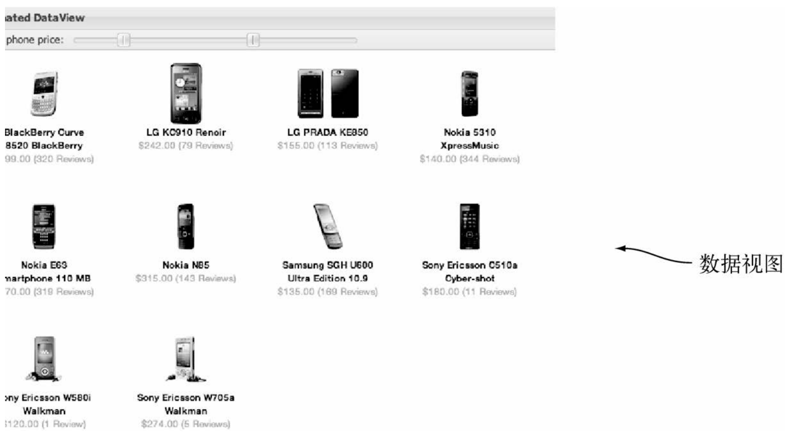


图1-8　数据视图在Ext JS SDK示例中的演示

`DataView`类从一个数据存储获取数据，用一个叫作`XTemplate`的类把它显示在屏幕上，并提供了一个简单的选择模型。Ext JS XTemplate是一个HTML片段生成工具，让你可以创造一个模板，并在里面为数据元素预留占位符，它可以用数据存储的独立记录填充，也可以在DOM上销毁。

列表视图部件不复存在！

如果你是Ext JS 3.0转用Ext JS 4.0，那么可能会奇怪列表视图部件哪儿去了。简言之，在Ext JS 3.0里提供能进行更快表格渲染的列表视图在Ext JS 4.0里被移除了，后者转而选择重构网格面板以大幅提高性能。

网格面板和数据视图是在屏幕上显示数据的基本工具，但它们有一个很大的限制，就是只能显示记录列表，不能显示分级数据。这时就需要树形面板来补缺了。

1.3.4 “枝繁叶茂”的树形面板

树形面板部件在众多显示数据的UI部件中是一个例外，因为它并非使用数据存储中的数据，而是通过使用`TreeStore`类来获取分级数据。图1-9展示了一个Ext JS树形面板部件的示例。在这里，树形面板被用于显示Ext JS框架安装目录内的父子数据。

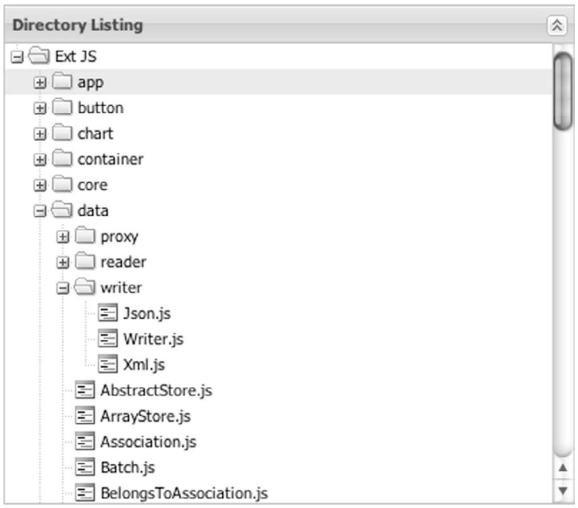


图1-9　一棵Ext JS树，它是源自Ext JS SDK的一个示例

在Ext JS 4.0中，它已经被彻底重建，现在是网格面板的近亲。图1-10展现了新树形面板的多功能性。

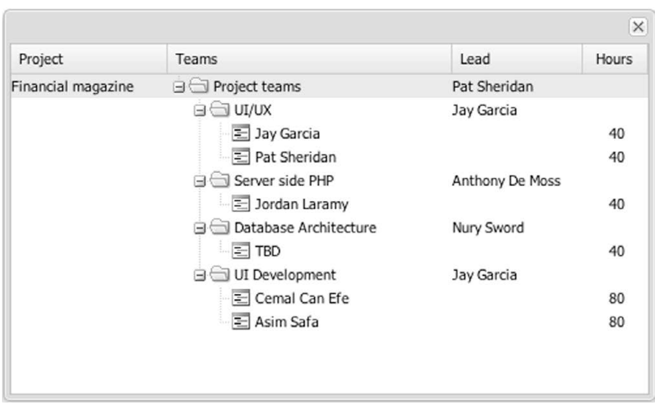


图1-10　一个有列的树形面板

在我们讨论容器的时候，你已经见过单行文本框（text field）了。接下来，我们要看看Ext JS提供的其他一些输入框。

1.3.5　表单输入框

Ext JS拥有8种输入框的控制板。有像之前已经提到的单行文本框这样的简单输入框，也有诸如ComboBox（组合框）和HTML编辑器之类复杂的输入框。图1-11展示了部分现成可以使用的Ext JS表单输入框部件。

在图1-11中可以看到，部分表单输入框看上去很像它们相应原生HTML输入框的风格化版本。不过，两者的相似性仅限于此。Ext JS表单输入框的功能远远不止它们表面那样！



图1-11　显示在一个封装窗口里的所有现成可用的表单元素

每一个Ext JS输入框（除了HTML编辑器）都包括一套工具，这些工具可以用来实施诸如取值赋值、标识输入框为无效、重置，以及执行输入框校验等操作。可以通过正则表达式或者自定义校验方法对输入框进行自定义校验，能够全盘掌控正在输入表单中的数据。输入框可以在数据被输入的同时校验数据，向用户提供实时反馈。

- 单行文本框和多行文本框

`TextField`（单行文本框）和`TextArea`（多行文本框）类可以被视为对其相应标准HTML输入框的扩展，拥有像校验之类的额外特性。`TextField`类是很多其他复杂部件——包括ComboBox（组合框）、Number field（数字输入框）和Time field（时间输入框）——的基础。

- 单选框和复选框

跟单行文本框一样，单选框和复选框是对原始HTML中单选框和复选框的扩展，不过它们包含了Ext JS元素管理的所有好处，而且有一些便捷类可以配合自动布局管理来创建复选框集合和单选框集合。图1-12展示了一个Ext JS中如何用复杂的布局来配置`CheckboxGroup`和`Radio-Group`类的小示例。

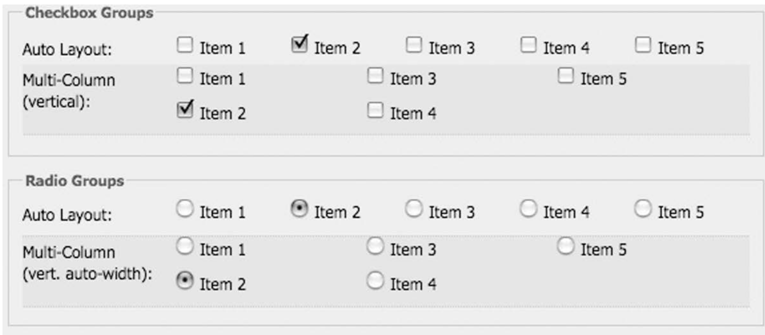


图1-12　一个`Checkbox`（复选框集合）和`RadioGroup`（单选框集合）便捷类与自动布局共同使用的示例

- HTML编辑器

HTML编辑器是所见即所得（What You See Is What You Get，WYSIWYG）的，就像是多行文本框的增强版。HTML编辑器使用现有的浏览器HTML编辑功能，在所有输入框当中可以被视为一个异类。关于这个输入框还有很多可以讨论的内容，我们将在第6章详述。但现在，让我们回到`ComboBox`和它的子类`TimeField`。

- `TriggerField`系列的输入框

`TriggerField`类是负责在单行文本框右侧显示一个按钮的基础类。它的子类被分成两组：拾取器和微调器。拾取器包含组合框和日期框，微调器包括微调区和数字区。

组合框绝对是最复杂和可配置性最强的表单输入框。它可以模拟传统的选项下拉框，还可以用远程数据集借助数据存储来设置。它可以被设置为自动完成用户输入的文本（也就是所谓的提前键入），并进行远程或者本地数据过滤。它也可以被设置为使用你自己创建的Ext JS XTemplate实例，在下拉框区域内显示一个自定义列表，也就是所谓的绑定列表（bound list）。图1-13是一个自定义组合框在实际使用中的示例，正被用于搜索Ext JS论坛。

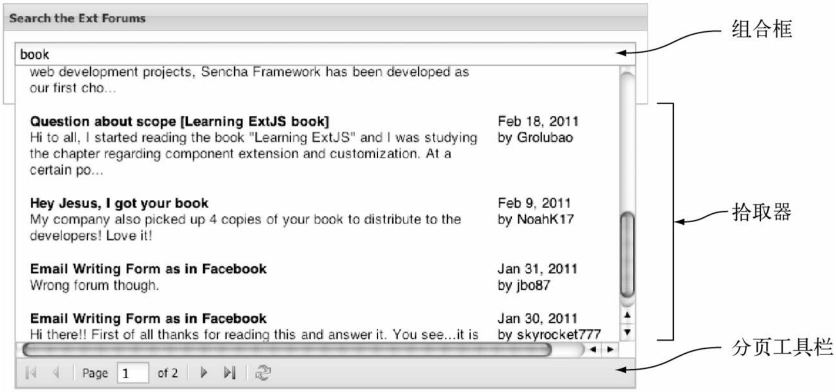


图1-13　一个自定义组合框，其中包含了一个集成分页工具栏，在可下载的Ext JS示例中可见

这里的组合框在列表框中显示网帖标题、日期、作者和网帖片段之类的信息。因为有些数据集范围非常大，所以它被设置为使用分页工具栏，以方便用户分页浏览结果数据。

因为组合框的可配置性很强，我们也可以在结果数据集中包含图像引用，将其显示在被渲染的结果数据中。

现在我们来到了UI之旅的最后一站，来看一看其他一些通用的UI组件。

1.3.6　其他部件

有一些很重要的UI控件，虽然不属于主要组件，但在UI的大框架中发挥着支持作用。在图1-14中可以看到屏幕上展现了一系列不同部件的控制面板。

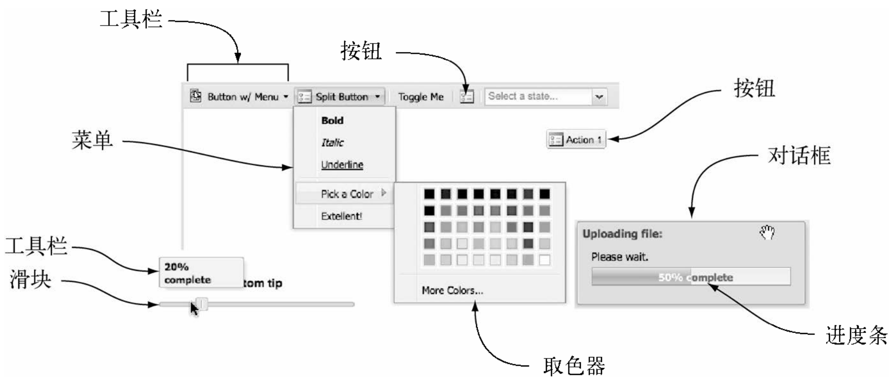


图1-14　UI部件和控件大杂烩

你已经了解到Ext JS是如何通过众多不同的部件完成工作的，也知道可以选用Ext JS来打造一个应用程序，而不用碰一星半点儿的HTML。你还从头到尾浏览了Ext JS框架，包括遍历它的UI。

目前讨论过的所有内容在Ext JS 3.0中都存在过。现在，让我们花点儿时间来了解Ext JS 4.0引入的新特性。

1.4 Ext JS 4.0 的新特性

若说Ext JS 4.0是JavaScript框架的一场革命时，我们绝没有夸大其词。这个框架有很多增强功能，有时候很难把握所有变化。这主要是因为很多改变之处都发生在表象层以下，位于Ext JS代码库的最深处，你很少会冒险尝试，因为它们有时候令人费解地复杂。

接下来，我们要看看这个框架发生的一些最鲜明的转变。如果有使用3.0版的经验，你可能会奇怪为什么框架的尺寸变大了，接下来的几节将告诉你原因。

1.4.1 呀！适配层不见了！

通过使用适配层，Ext JS 2.0和Ext JS 3.0可以凌驾于jQuery、Prototype和YUI库之上。而在Ext JS 4.0中，再也不是这样了。

虽然得到了从那些库转到Ext JS的开发人员的好评，但适配层一直都是争论的焦点，原因有多个。适配层的最主要问题一直都是基础库的版本会改变，从而导致Ext JS产生漏洞。

另一个广为人知的问题是框架命名空间冲突。Ext JS 1.0 ~ Ext JS 3.0通过向函数、字符串和数组原型中植入方法来扩展JavaScript。因为其他库也对类似的方法名称进行了同样的操作，所以Ext JS会破坏基础库所做的改动。

Sencha开发团队为防止与其他库产生此类冲突和紧张关系，把上述特性作为函数、字符串和数组单例转移到了`Ext.util`命名空间。做了这些改变后，Sencha团队决定取消适配层，让Ext JS和其他任何库一同工作，让你可以使用那些库的任何版本，而不用担心升级那些库会导致Ext JS代码出问题。

1.4.2 新的类系统

Ext JS 4.0配备了一个全新的类系统，其中包含诸如依赖注入和动态类加载之类的特性，在用Ext JS 4.0编写面向互联网的RIA时必不可少。

伴随动态类加载而来的是混入类（mixin）的概念，这是一种允许多重继承的现代面向对象编程模式。这使得Sencha开发团队在开发这一框架的时候更具创造力，在强化功能的同时减少了编码时间，有时还能提高某些类和部件的易用性。

了解混入类！

如果你对混入类这个编程概念感觉陌生，下面这篇文章可以很好地解释它：http://en.wikipedia.org/wiki/Mixin。

新的类系统虽然提供了很多新特性，但也带来了代价，那就是新的模式。在实例化或者定义一个类的时候，新的类系统提出了与Ext JS 3.0大相径庭的模式。虽然这些新模式可能会加大学习Ext JS 4.0的难度，但它们绝对会让你在编写应用代码时更有创造性。

说到类，Ext JS 4.0有一个完全重构的数据类系统，这就是我们接下来要讨论的内容。

1.4.3　数据包

Ext JS 4.0里这个全新的数据包，其根源可以追溯到Sencha Touch——用诸如模型之类的术语来替代记录。不过，Ext JS 4.0对数据包的改动所带来的功能和组织性能要远远超过Sencha Touch。

Ext JS 4.0数据包中整合了数量庞大的类，包括像本地存储代理和树状存储之类的新成员。本地存储代理允许数据使用浏览器的本地存储功能来存储和读取，而树状存储取代了Ext JS 3.0的树加载器，让你可以用树实现比以前多得多的功能。

数据包额外增加了一些特性，比如关联和校验，以及经过深思熟虑的功能性重组。图1-15展示了数据包的特性和功能是如何分割和联系的。虽然我们在后面会更详细地探讨这一点，但不妨在此透露一些细节以激起你的学习热情。

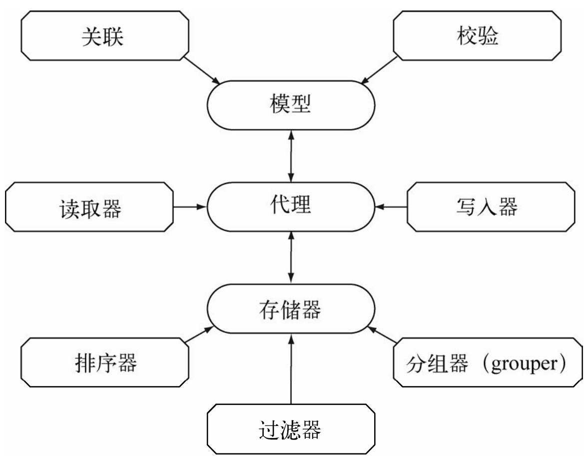


图1-15　Ext JS 4.0数据包

在Ext JS 4.0中，模型可以直接使用代理，而在之前版本的框架中则不能。与此类似，校验和关联现在也可以在模型层面执行。

数据包有了不少改动，而布局命名空间也同样经历了大幅度地重构。

1.4.4　布局：代码大爆炸

正如我们之前所讨论的，Ext JS 4.0拥有多达33种新的布局管理器，但其中只有13种是你需要了解的。这是因为布局被划分为两个主要功能区域：组件布局和容器布局。

组件布局和容器布局在这个框架里起到完全不同的作用。组件布局负责为组件组织HTML，而容器布局负责管理子组件的位置和大小。

既然我们谈到了布局的话题，那就让我们阐释一下Ext JS 4.0带给表格的新停靠系统。

1.4.5　新停靠系统

Ext JS的众多面板源自Sencha Touch，可以把像工具栏之类的部件停靠在所谓的*内容体区域*外侧，从而提高了这一部件的UI管理灵活性。如图1-16所示，三个工具栏分别停靠在一个面板的顶部、底部和左侧。而这在之前版本的Ext JS中，没有容器和布局的深层嵌套是不可能实现的。这一切都是通过称为Dock的组件布局来实现的。

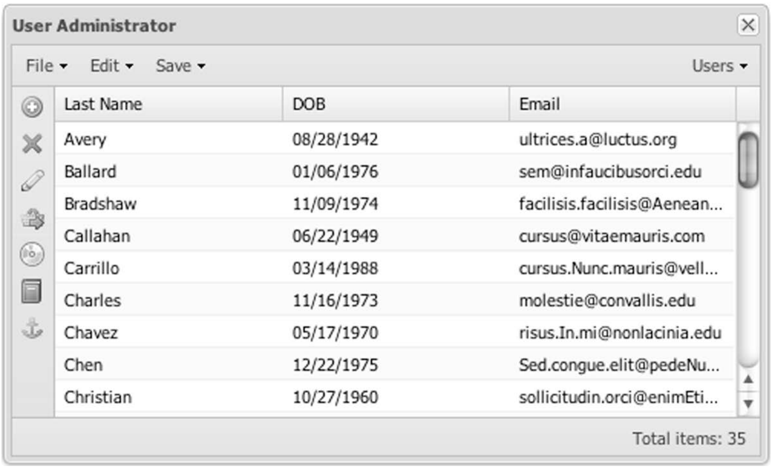


图1-16　演示Ext JS面板的新停靠特性

虽然可以期待充分利用Dock布局，但如果应用用到下面要讨论的网格面板，也会让你眼前一亮。

1.4.6　网格面板的改进

Sencha开发团队为了开发像网格面板这样的特性简直不分昼夜地工作，而回报是可喜的，尤其是在仔细看过从Ext JS 3.0到现在的变化以后。

网格面板的新特性中包含所谓的*无限网格*（infinite grid），使用它可以分页遍历大型数据集，而不需要包含分页工具栏。网格面板的其他新特性包括对命名空间进行了重新组织以更好地给类分组（参见图1-17）。

网格部分的代码被多组代码分割开来，其中包括列类型、插件和特性。虽然`data Store`从技术上来说并不在网格命名空间里，但它是`Grid panel`的一个支持类，所以我把它也放到了图1-17中。

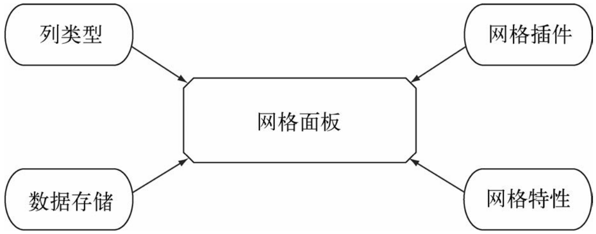


图1-17　网格面板支持的功能区域

网格包的这种组织水平意味着在配置面板时可以更灵活，使Ext JS可以只执行必须执行的代码。比如说想允许编辑网格单元，只需在网格面板配置中包含CellEditing插件。同样，如果想包含拖放功能的话，只需包含拖放插件。

其他功能被移植到所谓的特性命名空间（feature namespace），它跟插件有些类似，但是有区别。我不想在这里因讨论使用特性的细节把情况搞乱，不过有必要提出的是，像行分组或者提供数据汇总行之类的网格优势，可以只在你希望用的时候用。

正如你了解到的，网格面板的有了很大变动。但变动大的并不止它一个。接下来我们将看到，树面板也同样发生了一些重大的变化。

1.4.7　树形面板如今更接近网格面板

Ext JS树形面板的代码在Ext JS 1.0到Ext JS 3.0版一直都相对没有什么改动，但Ext JS 4.0的树形面板的代码被彻底重写。如果给网格面板和树形面板在之前的Ext JS版本中的差异画一个族谱图的话，要我说它们最多只是第三代堂表兄弟关系。而在Ext JS 4.0中，它们是兄弟关系！

如图1-18所示，网格面板和树形面板是兄弟关系，因为它们拥有同一个超类，也就是说它们拥有相同的基础代码。好消息是，一旦你掌握了它们其中之一，那么另一个学起来就会顺利很多。网格面板和树形面板拥有相同的基础代码，这意味着你可以在树视图中加入列之类的元素。

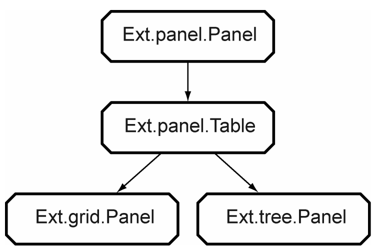


图1-18　树形面板和网格面板拥有同一个超类

也就是说，树形面板并不包含网格面板展示的很多功能，比如说汇总行插件或者列锁定。此外，树形面板必须使用数据包里的`TreeStore`类来管理和显示分级数据。

我们刚刚介绍了Ext JS框架自早期以来一直都有的两个主要的数据绑定视图。接下来，我们要面对的是全新的制图包。

1.4.8 图形和图表

图表最早是相对低调地被引入Ext JS 3.0中的。原因有两个：第一，这些图表是根据YUI库重新打包的基于Flash的图表；第二，对YUI图表包的升级经常会拖到几次版本修订之后，令开发人员颇感郁闷。

而在Ext JS 4.0中，YUI图表包被放弃了，转而分成两大块从头开始构建而成。第一大块是Ext Draw，这是Ext JS内部的一个迷你框架，其设计源于在Raphael JS项目中得到的经验。Raphael JS是一个运用矢量标记语言（VML）、可伸缩矢量图形（SVG）或者画布（Canvas）在浏览器中绘图的Sencha实验室项目。

第二大块是图表包，以Ext Draw为基础。新的图表包中包含了两种新图表：散点图和雷达图。图1-19展示的是雷达图（数据1、数据2、数据3分别以绿、红、蓝三色区分）。

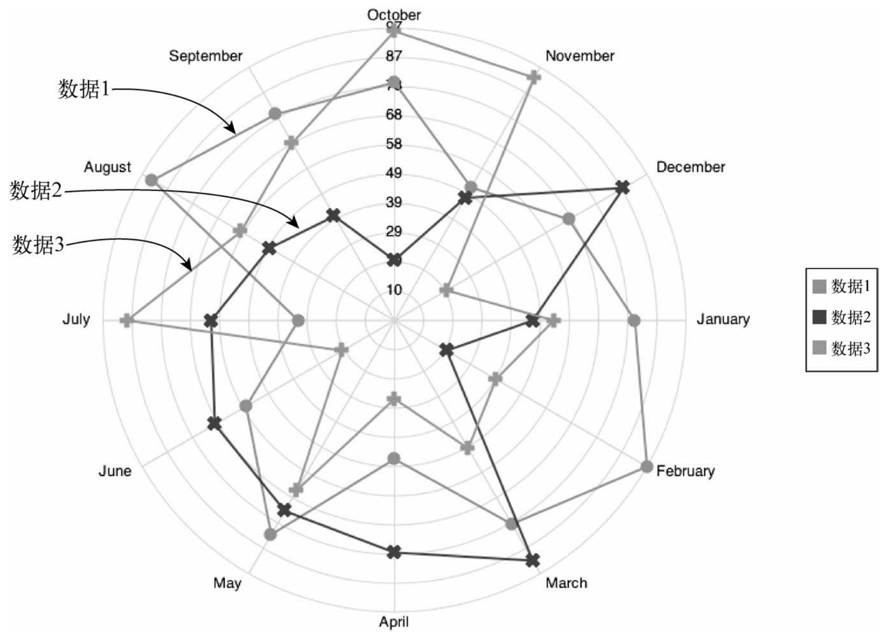


图1-19 Ext JS包含了一些不使用Flash的新图表

我们已经讨论过UI部件的很多元素，但除此之外还有一些深层次的内容值得在此提及。

1.4.9 新的CSS样式架构

Ext JS使用Sass让Sencha开发团队和用户都可以创建自定义主题。这就意味着如果你想改变整个配色方案，懂Sass可以让这一任务相对轻松地完成。

进一步了解Sass

Sass以迅雷不及掩耳之势席卷样式表管理领域，颠覆了人们给网页和应用制定样式的方式。要了解更多关于这一工具的情况，请查阅《Sass与Compass实战》（*Sass and Compass in Action*，Manning出版社，2013）。

借助自定义样式表和部件，可以用Ext JS开发应用。它们需要某个东西把它们联系在一起，而在Ext JS 4.0中，Sencha就提供了这样的工具。

1.4.10　新MVC架构

Ext JS一直缺乏的一样东西，就是一种使用该框架开发应用的可靠模式。而在Ext JS 4.0里不是这样了！根据从Sencha Touch中吸取的经验教训，Ext JS 4.0配备了一套可靠的MVC架构，让你可以使用久经考验的MVC模式来开发代码。我们将在本书的最后两章详细探讨这个问题。

Ext JS 4.0的新特性并不局限于浏览器内容。这个框架也配备了其他工具，可以在应用构建过程中使用。

1.4.11　捆绑打包工具

早些时候，我们了解到Ext JS 4.0配备了一个动态类加载系统。类加载器是一套针对基于互联网的Ext JS应用的优秀解决方案，但基于内联网的应用对于响应时间经常有着更高的要求，所以Sencha现在把它流行的JSBuilder打包压缩工具包括了进来，它就是用这个工具构建和打包Ext JS和Sencha Touch的！

我们花了很多时间了解Ext JS框架中的新内容，现在是时候下载并开始使用它了。

1.5　下载和配置

虽然下载Ext JS是一个简单的过程，但配置一个页面使之包含Ext JS，并不像在HTML中引用一个文件那么简单。现在你将了解它的配置、文件夹层次结构，以及到底有哪些文件夹和它们的用处。

你要做的第一件事是找到源代码，因此请访问www.sencha.com/products/ExtJS/download/。下载文件是zip格式的SDK，大小超过30 MB。稍后我会解释为什么这个文件这么大。现在，请把文件解压缩到你处理JavaScript文件的位置。要想使用Ajax，并且不用访问sencha.com就能查看文档，你需要一个Web服务器。我们通常用自己电脑上本地配置的Apache，它是免费和跨平台的，不过Windows的IIS也可以。

你可能会像我们一样查看从下载的软件开发工具包zip文件中解压缩出来的文件大小。如果你被看到的大小吓到了，放松点儿，把它装回去。是的，30 MB以上的大小对一个JavaScript框架来说非常大。现在，让我们先不管大小，看看图1-20展示的解压出的内容。

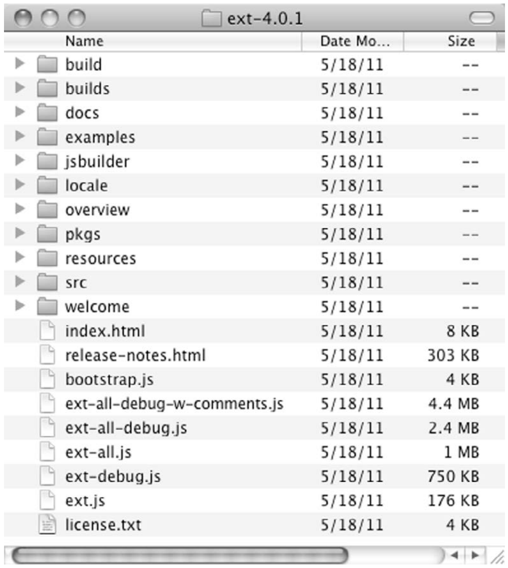


图1-20　Ext JS软件开发工具包内容一览

在软件开发工具包里，你可以看到很多内容。之所以有这么多文件夹和文件，这是因为下载包里包含整个代码库和CSS的多份副本。而这么做是为了让你能以自己认为合适的方式自由构建或者使用Ext JS。表1-1解释了其中每个文件夹各是什么以及各自的用途。

表1-1　Ext JS软件开发工具包的内容

文件夹	文件夹的用途
build	包含使用JSBuilder合并和压缩应用程序代码所必需的脚本
builds	包含Ext JS框架的三个不同版本。第一个是沙盒版，你可以在里面将Ext JS 3.0与Ext JS 4.0内联运行，以减少移植风险。第二个是库的核心，这个核心包含DOM管理和该框架里多个不同的工具。最后一个是Ext JS foundation，它是Ext JS框架的基础
docs、overview、welcome	docs（文档）文件夹包含了完整的API文档，而overview（概览）目录包含了该框架的简单介绍。welcome（欢迎）文件夹包含了支持框架启动界面的必要资源，双击index.html即可打开启动界面
examples	包含了所有示例源代码
jsbuilder	包含了JSBuilder的二进制文件和源代码
locale	包含45种口语翻译，以替代框架内的不同文本
pkgs	整个框架在这里被分解成压缩合并后的几大块，以供连接较慢的浏览器使用。它被分解为foundation、DOM、classes和extras。classes是目前为止最大的集合，其中包含所有的部件和数据存储代码，而extras文件夹包含有诸如JSON和Ext JS MVC应用等工具
resources	包含了CSS、图像和Sass源代码
ext*.js	在Ext JS的发布目录根目录下有多个不同的ext*.js文件。要知道任何名字里带“-debug”的文件都是该文件的非压缩版本。这些文件可以分为两组，第一组是ext.js和ext-debug.js，这两者包含了Ext JS框架的基础。当你想使用Ext JS类加载器的时候要包含它们。ext-all*文件是整个库文件打包成的一个文件。我们将用ext-all-debug来做练习

虽然在发布目录里有不少文件和文件夹，但只需其中的少数就能让该框架在浏览器中运行。现在就让我们来实际使用Ext JS。

1.6　亲手一试

在这个练习中，将创建一个Ext.form.Panel实例，它将在一个Ext.window.Window内部渲染。这个表单面板将包含两个文本输入框和一个按钮，按下后可以提供反馈。代码清单1-1展示了如何引导程序代码。

代码清单1-1　创建hello_world.html

```
<link rel="stylesheet" type="text/css"
    href="/ExtJS/resources/css/ext-all.css" />
<script type="text/javascript"
    src="/ExtJS/ext-all-debug.js"></script>
<script type="text/javascript" src='hello_world.js'>
</script>
```

代码清单1-1包含了一个典型的唯Ext JS设置的HTML标记，其中包含了合并的CSS文件ext-all.css、必需的JavaScript文件，以及ext-all-debug.js。最后，它还包含了即将创建的hello_world.js文件。

代码清单1-1用/ExtJS作为框架代码的绝对路径，如果你的路径不同一定要记得更改。创建一个指向hello_world.js文件的脚本标签，它将会包含你的主JavaScript代码。

现在你要分两步创建hello_world.js文件。第一步，如代码清单1-2所示，是构建表单面板和它相关的子组件。

代码清单1-2　创建hello_world.js

```
var tpl = Ext.create('Ext.Template', [          ❶ 创建一个Template实例
    'Hello {firstName} {lastName}!',
    ' Nice to meet you!'
]);
var formPanel = Ext.create('Ext.form.FormPanel', {     ❷ 配置表单面板
    itemId      : 'formPanel',
    frame       : true,
    layout      : 'anchor',
    defaultType : 'textfield',
    defaults    : {
        anchor     : '-10',
        labelWidth : 65
    },
    items       : [                              ❸ 设置两个输入框
        {
            fieldLabel : 'First name',
            name       : 'firstName'
        },
        {
            fieldLabel : 'Last name',
            name       : 'lastName'
```

```
            }
        ],
        buttons : [
            {
                text    : 'Submit',
                handler : function() {
                    var formPanel = this.up('#formPanel'),
                        vals      = formPanel.getValues(),
                        greeting  = tpl.apply(vals);
                    Ext.Msg.alert('Hello!', greeting);
                }
            }
        ]
    });
```

❹ 配置反馈按钮

❺ 显示Ext.Msg警告对话框

代码清单1-2包含了配置一个有两个输入框和一个按钮的表单面板所必需的代码。首先，创建一个`Ext.Template`❶实例，稍后将用它来创建一个动态对话框的内容主体。接下来，继续创建一个`Ext.form.FormPanel`❷实例，它包含两个文本输入框❸和一个按钮❹。按钮上配置了一个事件处理程序，用之前配置的Template和取自表单面板的值来显示一个`Ext.Msg`提示对话框❺。

“Hello world”示例差不多完成了，不过表单还没有渲染到屏幕上。为此，要调用`Ext.onReady`。在代码清单1-3中，需要把窗口中的表单面板封装起来以展现框架的灵活性。

代码清单1-3　全部组合在一起

```
    Ext.onReady(function() {

        Ext.create('Ext.window.Window', {
            height    : 125,
            width     : 200,
            closable  : false,
            title     : 'Input needed.',
            border    : false,
            layout    : 'fit',
            items     : formPanel
        }).show();

    });
```

❶ 调用`Ext.onReady`

❷ 渲染窗口

❸ 包含表单面板

代码清单1-3中包含了在一个Ext JS窗口内部渲染表单面板的代码。首先，它调用`Ext.onReady`❶，并代入一个匿名函数；当Ext JS判断浏览器已准备好操作DOM的时候，就执行这个函数。在这个匿名函数内部，创建了`Ext.window.Window`实例❷，它包含了`FormPanel`实例❸。图1-21展示了带有子表单面板的渲染示例。

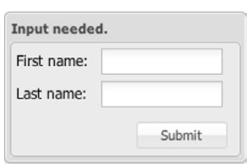


图1-21　渲染表单面板的示例窗口

图1-22展示了渲染在屏幕上的“Hello world”示例。为触发Submit（提交）按钮的事件处理程序，要在两个输入框里输入数据，然后单击Submit按钮。如果一切操作都正确的话，应该可以看到用在表单中输入的数据生成的`Ext.Msg`提示对话框。



图1-22　“Hello world”示例的最终结果

大功告成！你刚刚使用Ext JS在一个Ext JS窗口内部渲染了一个有相关输入框和一个按钮的表单面板。虽然这个示例本质上来说很简单，但确实展现了Ext JS的强大。

1.7　小结

在本章中，你学会了用Ext JS构建健壮的Web应用，知道了它相对于市面上其他流行框架具备的优势，它是唯一基于UI，包含像Component、Container和Layout模型这样以UI为中心的支持类的框架。

你探索了很多该框架提供的核心UI部件，并了解到很多预制部件有助于提高程序开发效率。我们还讨论了Ext JS 4.0做出的一些改变，比如全新的非Flash图表和MVC包。

最后，你看到了如何下载Ext JS框架，连同每个独立的基础框架一起设置该框架。你创建了一个“Hello world”示例，它展示了如何用一个Ext JS窗口来渲染带按钮的表单面板，并可以用一些简单的JavaScript呈现一个`Ext.Msg`提示对话框。

在接下来的几章中，你将由内而外地探索Ext JS的工作机制。这些知识将帮助你在创建结构优良的UI时做出最好的决定，并提高你有效使用该框架的能力。这将是一次有趣的旅程。

第 2 章

DOM操作

2

本章内容

- ❑ 引导JavaScript代码
- ❑ 用Ext.Element管理DOM元素
- ❑ 通过Ajax加载HTML片段
- ❑ 在一个HTML元素上实现高亮效果
- ❑ 实现模板和XTemplate

开发应用的时候，你可能会通过类比进行思考。比如说，我们喜欢把一个应用的启动时间控制类比航天飞机的发射，后者的时间控制可以决定发射成败。在处理任何操作DOM的程序时，明确何时初始化JavaScript代码至关重要。本章里，你将了解如何用Ext JS启动自己的JavaScript，以确保应用代码在每种浏览器上都能在恰当的时间初始化。然后，我们将探讨如何使用Ext.Element操作DOM。

众所周知，DOM操作是网页开发人员大多数时候受命编程的任务之一。不管是添加还是移除DOM元素，我敢肯定你感受过用“开箱即用”的JavaScript方法来履行这些任务有多痛苦。毕竟，DHTML（Dynamic Hyper Text Makeup Language，动态超文本标记语言）长期以来一直是动态网页的核心技术。

我们将审视Ext JS的核心，即Ext.Element类，它是一个健壮可靠的、跨浏览器的DOM元素管理套件。你讲学习如何利用Ext.Element对DOM添加节点和移除节点，而且也将看到这会让任务完成得更轻松。

一旦熟悉了Ext.Element，你将学习如何使用模板在DOM之中生成HTML片段。我们还将深入探究XTemplate的使用，介绍如何用它轻松地遍历数据，并在同时插入行为修正逻辑。这会是很有趣的一章。在写代码以前，请一定先了解一下如何引导启用了Ext JS的Web应用。

2.1 用 Ext JS 启动代码

大多数开发者从早期开始就是这么初始化JavaScript的，即在要加载的HTML页面中给<body>标签添加一个onLoad属性：

```
<body onLoad="initMyApp();">;
```

虽然这种调用JavaScript的方法可行，但对于启用Ajax的Web 2.0网站或者应用来说并不理想，因为onLoad代码通常在不同的浏览器上触发的时间不一样。比如说，有些浏览器在DOM准备好，以及所有内容都已被浏览器加载渲染好之后触发这一方法。而对于Web 2.0而言，这不是好事，因为代码通常偏好在DOM准备好，但还没有任何图像加载以前，就开始管理和操作DOM元素。这时你可以实现触发时机和性能之间的完美平衡。我们喜欢称之为页面加载周期中的“甜区”。

和浏览器开发领域的很多东西一样，每个浏览器通常都有一套自己特有的方式，来探知它的DOM节点何时可以进行操作。

原生浏览器解决方案可以侦测DOM是否就绪，但它们在不同浏览器上的实现不尽相同。比如说，Firefox和Opera触发DOMContentLoaded事件。Internet Explorer要求在文件中置入一个带defer属性的脚本标签，在DOM就绪的时候触发。WebKit并不触发事件，但会把document.readyState属性设置为complete，所以必须要执行一个循环检查该属性，并触发一个自定义事件来通知代码DOM已就绪。好家伙，太乱了！

幸运的是有Ext.onReady，它可以解决触发时间问题，并作为启动应用特定代码的基础。Ext JS通过侦测代码在哪种浏览器上执行，以及管理对DOM就绪状态的侦测，来实现跨浏览器兼容，在恰当的时间执行代码。

Ext.onReady是Ext.EventManager.onDocumentReady的一个引用，并接受三个参数：调用的方法、从中调用方法的作用域，以及传给该方法的任何选项。第二个参数，即scope，是当你调用一个需要在特定作用域里执行的初始化方法时使用的。

了解一下作用域

很多JavaScript开发人员在他们职业生涯早期经常纠结于作用域的概念，而这个概念是每名JavaScript开发人员都应该掌握的。你可以在以下网址找到学习了解作用域的优质资源：www.digital-web.com/articles/scope_in_javascript/。

所有基于Ext JS的JavaScript代码都出现在Ext JS脚本的引入以下（以后）。这种配置很重要，因为JavaScript文件是被同步请求和加载的。在Ext JS命名空间中定义以前就试图调用任何Ext JS方法都会导致异常，代码将载入失败。这是一个使用Ext.onReady触发Ext JS MessageBox警告对话框的示例：

```
Ext.onReady(function() {
    Ext.Msg.alert('Hello', 'The DOM is ready!');
});
```

在上述的示例中，把一个所谓的匿名函数传递给Ext.onReady作为唯一参数，当DOM做好操作准备的时候执行该函数。匿名函数中包含一行调用Ext JS MessageBox的代码，如图2-1所示。



图2-1　Ext.onReady调用的结果，显示一个Ext.MessageBox窗口

匿名函数是指那些没有变量引用的函数，或者在对象中通过属性键引用的函数。Ext.onReady注册了匿名函数，当内部事件docReadyEvent被触发时执行该函数。简而言之，一个事件就像是一条通知某事已发生的信息。事件监听器（listener）是一个方法，注册用于在该事件发生或触发时进行执行或调用。

当Ext JS发现在页面加载周期中执行匿名方法或者其他已注册监听器的最佳时间时（还记得“甜区”吧），就会触发这个docReadyEvent事件。如果你觉得事件这个概念听起来有点让人困惑，那也别慌。事件管理是一个复杂的话题，我们将在第3章讲到。

使用Ext.onReady的重要性再怎么强调也不为过，所有示例代码都*必须要*这样载入。在后文中，如果Ext.onReady没有在示例中详细写明，请默认用它来载入代码，并以如下方式封装示例代码：

```
Ext.onReady(function() {
    //……代码在此……
});
```

你已经很清楚如何用Ext.onReady来载入代码，现在花点儿时间了解一下Ext.Element类。这方面的知识非常必要，在框架中有DOM操作的所有地方都会用到。

2.2　用 Ext.Element 管理 DOM 元素

所有基于JavaScript的Web应用都是围绕着一个中心，那就是HTML Element（元素）。JavaScript对DOM节点的可操作性使你有这个能力和灵活性对希望操作的DOM进行任何操作，这些操作包括添加、删除、编辑样式，或者改变文件中任何节点的内容。通过ID引用DOM节点的传统方法如下：

```
var myDiv = document.getElementById('someDivId');
```

使用getElementById方法可以执行一些基本任务，比如更改innerHTML，或者编辑一个CSS类的样式并分配这个类。但如果想对节点进行更多操作呢？比如管理它的事件，给鼠标击键应用一个样式，或者替换单个CSS类，那就必须管理自己编写的所有代码并持续更新，以确保代码能百分百地跨浏览器兼容。而这是我们最不想在上面多花时间的差事。幸好，Ext JS为你代劳了。

2.2.1　框架的核心

让我们来看Ext.Element类，它被众多Ext JS开发人员公认为是该框架的核心，因为它在所

有UI部件中都起到一定的作用，而且通常可以通过`getEl`方法或者`el`属性访问。

`Ext.Element`类是一个完整的DOM元素管理套件，其中包含了众多有用的工具，使该框架可以对DOM实施各种操作，并提供了我们逐渐青睐的健壮可靠的UI。这套工具集及其全部功能都可以由你，即最终开发者运用。

因为它的设计，它的功能并非对DOM元素的简单管理，而是用于执行一些复杂任务，比如相对方便地管理维度，对齐和坐标。你还可以轻松地利用Ajax更新一个元素，管理子节点，实现动画，享受完整的事件管理，其他好处还有很多。

2.2.2　首次使用`Ext.Element`

`Ext.Element`很易于使用，而且可以把一些极难的任务变简单。要利用好`Ext.Element`，需要设置一个基页。如我们在第1章中所讨论的，设置一个页面，在其中包含Ext JS和CSS。然后，包含以下CSS和HTML。

```
<style type="text/css">
    .myDiv {
        border: 1px solid #AAAAAA;
        width:  200px;
        height: 35px;
        cursor:  pointer;
        padding: 2px 2px 2px 2px;
        margin: 2px  2px  2px  2px;
    }
</style>
<div id='div1' class='myDiv'></div>
```

借助这段代码，确保了目标`div`标签都有特定的维度，并设置了一个边框，在页面上可以清晰地看到它们，从而为这本书中的示例架设好了平台。包含了一个`id`为`'div1'`的`div`，之后将用它来作定位。如果正确地设置了页面，风格化的`div`应该如图2-2显示的那样清晰可见。这张图展现了通用的HTML盒，我们将用它来练习基本的`Ext.Element`方法。

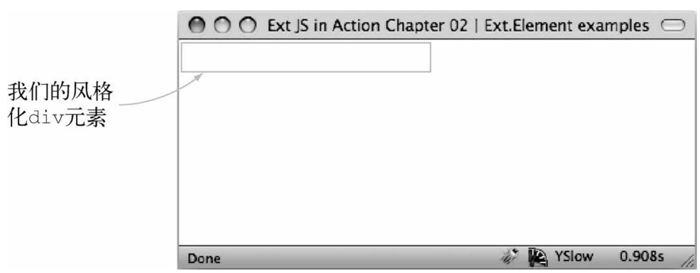


图2-2　你的基本页面，已准备就绪进行Ext JS `Element`操作的风格化`div`

注意 所有的Ext.Element示例代码都将引用刚刚设置的基页。如果有兴趣实时观看对DOM所做的变更，我们建议你使用Firefox里多行的Firebug文本编辑器来编辑这些示例。如果你对Firebug不熟悉，可以访问http://getfirebug.com/wiki加以了解。或者你也可以把这些示例置入通用代码块。只是记得要使用Ext.onReady。

CSS中规定，任何与类myDiv相关联的div都被设定为高35像素和宽200像素，看起来有点奇怪。让我们把该元素的高度设为200像素，使之成为正方形。

```
var myDiv1 = Ext.get('div1');
myDiv1.setHeight(200);
```

上面这两行代码的执行很重要。第一行使用Ext.get方法，向它传递一个为字符串'div1'，而返回一个Ext.Element类的实例，用变量myDiv1来引用。Ext.get方法用到document.getElementById，并用Ext JS的元素管理方法对其进行封装。

使用新引用的Ext.Element实例myDiv1，并调用其setHeight方法，传入一个整数值200，使之框架高度增加到200像素。或者，也可以用它的setWidth方法改变该元素的宽度，不过我们跳过这一步来玩一些更有趣的。

“现在变成正方形了。那有什么了不起？”你可能会说。假设再来改变一下它的尺寸，这一次是用setSize方法。把width和height设为350像素。用已经创建好的引用变量myDiv1:

```
myDiv1.setSize(350, 350, {duration:1, easing:'bounceOut'});
```

当执行这一行代码会如何？它会生动地呈现弹跳的效果吗？那样更好！

本质上，setSize方法就是setHeight和setWidth的合成。通过这个方法传入目标宽度和高度，以及一个有两个属性duration和easing的对象。而第三个属性，如果定义过，将使得setSize方法以动画形式呈现该元素的尺寸变化过程。如果不要动画，那就忽略第三个参数，该框架将立即变换尺寸，就像设置高度的时候。

尺寸设置是用Element类进行元素管理的众多方面之一。Ext.Element部分最强的功能源自其对元素CRUD（创建、读取、更新和删除）全套操作的易用性。

2.2.3　创建子节点

JavaScript的重要功能之一是它对于DOM的操作能力，这包含了对DOM节点的创建。JavaScript提供了很多本地方法来支持这一功能。Ext JS用Ext.Element类合适地封装了很多这些方法。让我们来创建几个子节点玩玩。

要创建子节点，使用Element的CreateChild方法：

```
var myDiv1 = Ext.get('div1');
myDiv1.createChild('Child from a string');
```

这段代码在目标div的innerHTML上添加了一个字符节点。如果想创建一个元素呢？易如反掌。

```
myDiv1.createChild('<div>Element from a string</div>');
```

对CreateChild的这种使用可以在div1的innerHTML上添加一个带字符串'Element from a string'的子div。我们不喜欢用这种方式添加子节点，因为觉得这样元素的字符串表现形式太过凌乱。Ext JS通过接受一个配置对象而非字符串，来解决这个问题：

```
myDiv1.createChild({
    tag  : 'div',
    html : 'Child from a config object'
});
```

在这里，使用一个配置对象创建一个子元素。指定tag属性为'div'，html属性为一个字符串。严格来说，这个跟之前用createChild来实现效果相同，但代码更整洁，而且自文档化。如果想插入嵌套标签呢？通过配置对象的这个方法，可以轻松实现：

```
myDiv1.createChild({
    tag      :'div',
    id       :'nestedDiv',
    style    :'border:1px dashed; padding:5px;',
    children :{
        tag   :'div',
        html  :'...a nested div',
        style :'color:#EE0000; border:1px solid'
    }
});
```

在这段代码中，创建了最后一个子节点，设置了id，应用了一定样式，还创建了一个子元素，该子元素是一个应用更多样式的div。图2-3展现了对div所做的改变是什么样。可以看到对myDiv1所做的所有添加，包括Firebug的实时DOM视图，上面显示添加了一个字符串节点和三个子div，其中一个子div有它自己的子div。

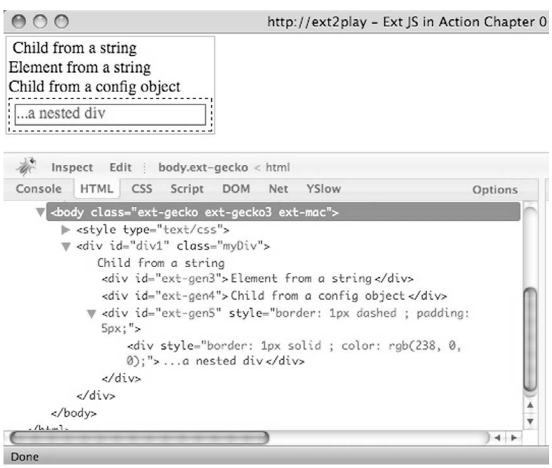


图2-3 用myDiv1.createChild所做的元素添加操作的集成

如果想在列表的顶端插入一个子节点，就使用便捷方法insertFirst，比如这样：

```
myDiv1.insertFirst({
    tag  :'div',
    html :'Child inserted as node 0 of myDiv1'
});
```

Element.insertFirst将总是在位置0插入一个新元素，即使在DOM结构中并不存在子元素。

如果想把一个子节点插入一个特定的索引位置，那createChild方法可以完成这个任务。唯一需要做的就是传入关于新创建节点插入位置的引用，像这样：

```
myDiv1.createChild({
    tag  : 'div',
    id   : 'removeMeLater',
    html : 'Child inserted as node 2 of myDiv1'
}, myDiv1.dom.childNodes[3]);
```

在这段代码中，向createChild方法中引入两个参数。第一个是新创建DOM元素的配置对象表示，而第二个是对目标子节点的DOM引用，而该目标子节点将作为createChild方法插入新创建节点的目标。请记住为这个新创建节点设置的id，稍后将会用到。

注意，这里用到了myDiv1.dom.childNodes。借助Ext.Element的dom属性，可以使用所有通用浏览器元素管理功能。

注意 Element.dom属性与document.getElementById()返回DOM对象引用相同。

图2-4展现了插入的节点在页面视图中的外观，和在Firebug DOM检验工具所呈现的DOM继承关系中的形态。可以看到，节点检验功能运行正常。用insertFirst方法在列表的顶端插入一个新节点，并用createChild方法在子节点3上方插入一个节点。当给子节点计数时要始终牢记，从数字0而不是1开始计数。

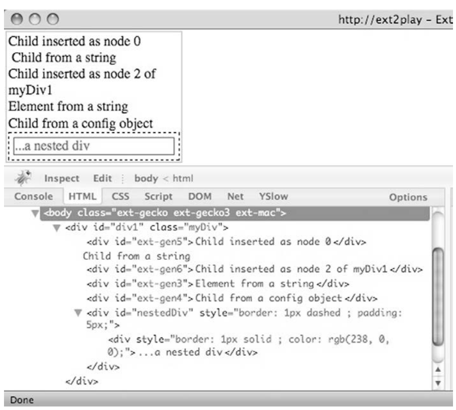


图2-4 使用一个索引位置配合createChild方法，并使用insertFirst方法，进行目标DOM元素插入操作的结果

“添加”是网页开发人员经常用到的操作。毕竟，这就是DHTML的部分意义所在。但知道如何删除也同等重要。让我们来看看如何用Ext.Element删除一些子元素。

2.2.4　删除子节点

删除节点比添加节点要容易得多，所要做的就是用Ext JS定位该节点，并调用其remove方法。为了测试对子节点的删除，首先需要一张干净可控的页面以供测试。用以下HTML创建一个新页面：

```
<div id='div1' class="myDiv">
    <div id='child1'<Child 1</div>
    <div class='child2'>Child 2</div>
    <div class='child3'>Child 3</div>
    <div id='child4'>Child 4 </div>
    <div>Child 5</div>
</div>
```

细看这段HTML，会发现一个id为'div1'的父div。它有5个直接后裔，其中第一个的id是'child1'。第二和第三个子节点没有id，但它们有CSS类'child2'和'child3'。第四个子元素id是'child4'，并有一个CSS类'sameClass'。类似的，它有一个直接子节点，id是"nestChild1"，跟它的父元素拥有相同的CSS类。div1的最后一个子节点没有id也没有CSS类。之所以要有这些设置，是因为要开始使用CSS选择器来定位，或者直接定位不同元素的id。

在添加子节点的示例中，一直都是把父div（id='div1'）封装在一个Ext.Element类里来引用这个父div，并使用它的create方法。而要删除一个子节点，办法就不一样了，需要明确定位需要被删除的节点。让我们使用这个新的DOM结构，来练习几种实现方法。

从一个已经封装的DOM元素上删除一个子节点的第一种实现方法。创建一个Ext.Element的实例，封装div1，然后用它借助于一个CSS选择器找到它的第一个子节点：

```
var myDiv1 = Ext.get('div1');
var firstChild = myDiv1.down('div:first-child');
firstChild.remove();
```

在这个示例中，用Ext.get创建了一个对div1的引用。然后用Element.down方法创建了另一个对首个子节点的引用，firstChild。传入一个伪类选择器，这导致Ext JS在DOM树中的div1前后询问第一个子节点，也就是一个div，并将其封装在Ext.Element的实例中。

Element.down方法在第一层的DOM节点中查找任何Ext.Element。正巧查找到的元素是id为'child1'的div。然后调用firstChild.remove方法，从DOM中删除该节点。

用选取器从列表中删除最后一个子节点的代码如下：

```
var myDiv1 = Ext.get('div1');
var lastChild = myDiv1.down('div:last-child');
lastChild.remove();
```

这个示例跟前一个原理很相似。最大的差别是用了选取器'div:last-child'，它定位了

div1的最后一个childNode，并将其封装在一个Ext.Element的实例中。在那之后，调用lastChild.remove方法，它就被删除了。

注意 CSS选择器是在DOM中查找项目的强大工具。Ext JS支持CSS3选择器规范。如果是刚刚接触CSS选择器，我们建议你访问W3C网页（http://mng.bz/0vmd），上面有关于选择器的大量信息。

如果想通过id来定位一个元素呢？可以用Ext.get来代劳。这一次，不用创建任何引用，而是用链接（chaining）来完成工作。

```
Ext.get('child4').remove();
```

执行这行代码可以删除id为'child4'的子节点及其子节点。始终记住：删除一个有子节点的节点也将删除其子节点。

注意 如果你想阅读更多关于链接的信息，业内领先的开发者Dustin Diaz在他的网站上有一篇很不错的文章，请访问：www.dustindiaz.com/javascript-chaining/。

我们要看的最后一项任务是用Ext.Element执行Ajax请求，以从服务器加载远程HTML片段并将它们注入DOM。

2.2.5 配合**Ext.Element**使用Ajax

Ext.Element类有能力执行Ajax调用，以取回远程HTML片段，并将那些片段插入其innerHTML。需要先写一个HTML代码段来加载它：

```
<div>
    Hello there! This is an HTML fragment.
    <script type="text/javascript">
        Ext.getBody().highlight();
    </script>
</div>
```

在这个HTML片段中，有一个简单的div，其中含有一个内嵌脚本标签，执行一个Ext.getBodycall方法。它使用链接来执行该调用的结果，执行它的highlight方法。Ext.getBody是一个便捷方法，以获得对Ext.Element封装的document.body的引用。把这个文件存为htmlFragment.html。

然后，要执行以下代码段的载入：

```
Ext.getBody().load({
    url     : 'htmlFragment.html',
    scripts : true
});
```

在这个代码段中，调用`Ext.getBody`调用执行结果的`load`方法，传入一个详细列明所要访问`url`（也就是htmlFragment.html文件）的配置对象，并把`scripts`设置为`true`。当执行这段代码，会出现什么样的结果呢？参见图2-5。



图2-5　往文档主体中加载一个HTML片段

执行这个代码段时，可以看到文档主体执行了一个取回htmlFragment.html文件的Ajax请求。在该文件被取回的同时，显示一个加载进度条。一旦该请求完成，HTML片段就被插入DOM。然后整个主体元素显示黄色高亮，这意味着JavaScript得到了执行。现在已经看到，使用`Ext.Element.load`工具方法与手工编码`Ext.Ajax.request`调用相比非常便捷。

就是这样。对DOM进行的添加元素和删除元素操作用`Ext.Element`很容易实现。Ext JS还有另一种添加元素的方法甚至比这还要简单，尤其是要把可重复的DOM结构置入DOM的时候。我们接下来要探讨`Template`和`XTemplate`实用工具类。

2.3　使用模板和 XTemplate

`Ext.Template`类是一个强大的核心工具，可以用来创建完整的DOM层级，其中留有空位以备之后用数据填充。当定义了一个模板时，可以用它来复制一个或者多个预定义DOM结构，并将数据填入空位。掌握模板可以帮助你掌握使用模板的UI部件，比如网格面板、数据视图和ComboBox（组合框）。

2.3.1　使用模板

首先要创建一个非常简单的模板，然后再进一步创建一个复杂得多的模板：

```
var myTpl = Ext.create('Ext.Template' , "<div>Hello {0}.</div>");
myTpl.append(document.body, ['Marjan']);
myTpl.append(document.body, ['Michael']);
myTpl.append(document.body, ['Sebastian']);
```

在这个示例中，创建一个`Ext.Template`的实例，然后传入一个`div`的字符串表示和一个空位，空位用花括号标示。然后在变量`myTpl`里存储一个引用。再后调用`myTp.append`，并传入一个目标元素`document.body`，以及用于填充空位的数据，这里的数据碰巧就是一个包含有姓氏的单元素数组。

连续如此操作三次，三个div就正好被添加进了DOM，每个空位中都填充了一个不同的姓氏。图2-6展现了调用append后产生的结果。



图2-6 用第一个模板往DOM中添加节点，如Firebug的分解图所示

可以看到，三个div被添加在文档主体上，每个都有不同的名字。使用模板的好处应该很清楚了。只要设置一次模板，就可以用不同的值把它套用到DOM上。

在之前的示例中，空位都是用花括号括起来的整数，而后传入单元素数组。模板也可以用从纯对象映射对象的键或值。以下代码清单展现了如何用这种语法创建一个模板。

代码清单2-1 创建一个复杂的模板

```
var myTpl = Ext.create('Ext.Template', [                              ❶ 创建复杂模板
    '<div style="background-color : {color}; margin:10px;">',
        '<b> Name : </b> {name}<br />',
        '<b> Age  : </b> {age}<br />',
        '<b> DOB  : </b> {dob}<br />',
    '</div>'
]);

myTpl.compile();                                                      ❷ 编译模板以加快速度

myTpl.append(document.body,{                                          ❸ 添加模板到文档主体
    color :"#E9E9FF",
    name  :'John Smith',
    age   :20,
    dob   :'10/20/89'
});

myTpl.append(document.body,{
    color :"#FFE9E9",
    name  :'Naomi White',
    age   :25,
    dob   :'03/17/84'
});
```

当我们创建这个复杂模板❶的时候，你注意到的第一件事或许就是传入了好几个参数。这是因为在创建一个模板时看以制表符分隔格式的伪HTML要比看长字符串容易得多。Ext JS开发人员喜欢这个主意，所以他们编写了`Template`构造函数来读取所有传入的参数，不管有多少个参数。

在`Template`伪HTML中，为4个数据点预留了空位。第一个是`color`，将被用来定义元素背景的式样。其余三个数据点是`name`、`age`和`dob`，当模式被添加上去的时候会直接显示出来。

下一步是编译❷模板，这样可以通过去除上面的正则表达式，加快模板合并数据和HTML片段的速度。对于这两项操作，严格来说并不需要编译，因为根本看不出速度上的提升，但对于使用众多模板的更大应用来说，编译的好处显而易见。保险起见，我们总是在实例化模板之后就编译它们。

最后，进行了两次`append`调用❸，并借此传入引用元素和一个数据对象。这次不像第一个模板示例中那样传入一个数字，而是传入一个数据对象，其中的键与模板的空位匹配。图2-7通过Firebug中的DOM视图展现了该复杂模板的执行结果。

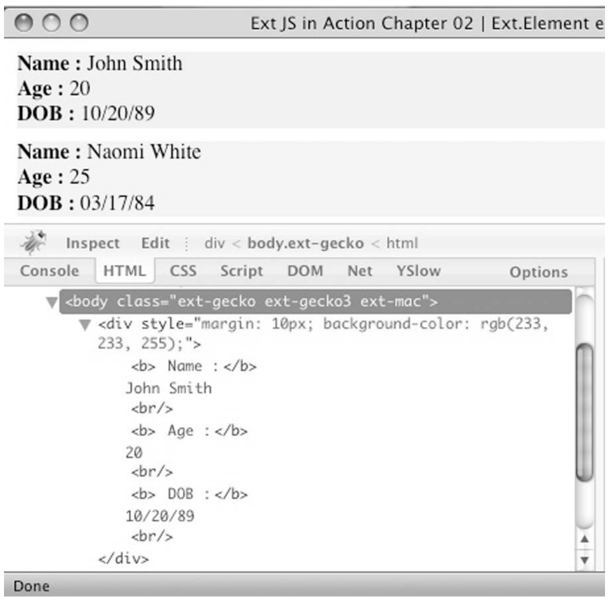


图2-7　该复杂模板在Firebug的DOM视图中的执行结果

使用模板，可以获得DOM中两个式样不同的HTML结构。可如果有一个对象数组呢？比如说，如果一个Ajax请求返回一个数据对象数组，而需要给每个数据对象应用一个模板该怎么办？一种解决方法是对数组进行遍历循环访问，这借助于通用的`for`循环或者更健壮的`Ext.each`工具方法可以轻松实现。我不会采用这种方法。我更倾向于使用XTemplate代替，它让代码更整洁。

2.3.2 用XTemplate执行循环操作

XTemplate严格意义上说可以被用于单数据对象，但它最有用的地方，是当必须要循环访问数组数据生成HTML片段的时候，用它要方便得多。XTemplate类扩展了Template类，并提供了许多额外功能。为了探讨XTemplate，首先要创建一个数据对象数组，然后创建一个XTemplate，用来生成HTML片段，如果下面的代码清单所示。

代码清单2-2 使用XTemplate来循环访问数据

```
var tplData = [{                                    ❶ 添加数据
    color : "#FFE9E9",
    name  : 'Naomi White',
    age   : 25,
    dob   : '03/17/84',
    cars  : ['Jetta', 'Camry', 'S2000']
},{
    color : "#E9E9FF",
    name  : 'John Smith',
    age   : 20,
    dob   : '10/20/89',
    cars  : ['Civic', 'Accord', 'Camry']
}];
                                                    ❷ 实例化新的
var myTpl = Ext.create('Ext.XTemplate', [              XTemplate
    '<tpl for=".">',
        '<div style="background-color:{color}; margin:10px;">',
            '<b> Name :</b> {name}<br />',
            '<b> Age :</b> {age}<br />',
            '<b> DOB :</b> {dob}<br />',
        '</div>',
    '</tpl>'
]);

myTpl.compile();                                    ❸ 添加HTML片段
myTpl.append(document.body, tplData);
```

在代码清单2-2中，首先设置了一个数据对象数组❶，它类似于在上个模板示例中使用的数据对象，只是额外增加了一个cars数组，在下一个示例中将会用到。

接下来，创建一个XTemplate实例❷，它的配置看起来很像之前那个Template，只是用一个自定义tpl元素封装了div容器，而tpl元素带属性for，并赋值为"."❸。tpl标签就像是模板的一个逻辑或者行为修饰符，它有两个算符for和if，用于更改XTemplate生成HTML片段的方式。在这里，赋值"."就是指示XTemplate在数组的根节点中循环查找传入它的值，并根据封装在tpl元素内的伪HTML构造HTML片段。当查看渲染好的HTML时，不会看到有tpl标签渲染在DOM上。采用这种方式跟模板示例的运行结果相同，如图2-8所示。

记住，这里使用XTemplate的好处在于不用写代码来循环访问对象数组，这种累活只需要让框架来代劳。XTemplate的功能远不止是循环访问数组，而这也极大地提高了它的可用性。

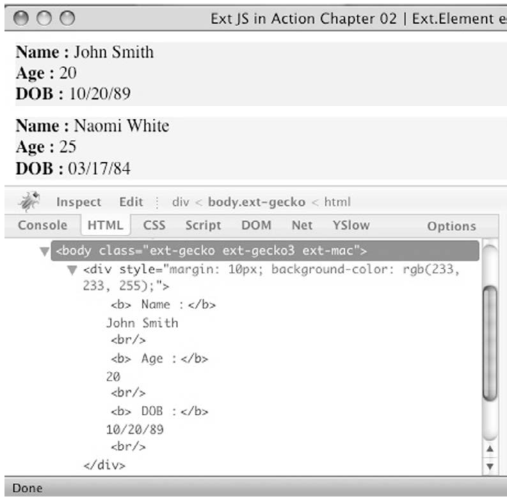


图2-8　通过Firebug的分解DOM视图使用XTemplate的结果

2.3.3　XTemplate的高阶应用

可以配置XTemplate，让它循环访问数组内的数组，甚至还可以有条件逻辑。下面这个代码清单中的示例将充分发挥XTemplate的一些功能，并展示很多此类高阶概念。你对即将看到的部分语法可能很陌生，但请不要气馁，我们将会逐一解释。在这个XTemplate的高阶应用中，我们要用到之前代码清单2-2中的`tplData`。

代码清单2-3　XTemplate的高阶应用

```
var myTpl = Ext.create('Ext.XTemplate', [
    '<tpl for=".">',
        '<div style="background-color:{color}; margin:10px;">',
            '<b> Name :</b> {name}<br />',
            '<b> Age :</b> {age}<br />',
            '<b> DOB :</b> {dob}<br />',
            '<b> Cars :</b>',
            '<tpl for="cars">',                          ❶ 循环访问cars数据
                '{.}',                                   ❷ 显示当前数据
                '<tpl if="this.isCamry(values)">',       ❸ 执行this.isCamry方法
                  '<b> (same car)</b>',
                '</tpl>',
                '{[ (xindex < xcount) ?", " :"" ]}',     ❹ 检验是否是数组末尾
            '</tpl>',
            '<br />',
            '</div>',
     '</tpl>',
    {
       isCamry : function(car) {                         ❺ 添加方法
           return car === 'Camry';
    }
```

```
        }
    ]);

    myTpl.compile();
    myTpl.append(document.body, tplData);
```

这段对于XTemplate的使用展现了好几个高阶概念，首先是在数组内部循环访问❶。记住，for属性指示XTemplate循环访问一个值列表。在这里，for属性的值是'cars'，不同于第一个for属性赋的值"."。这个属性指示XTemplate循环访问这一块伪HTML代码查找每辆车。记住，cars是一个字符串数组。

在这个循环内是一个带"{.}"的字符串❷，它指示XTemplate把数组的值赋给该循环的当前索引。简而言之，一辆车的名字将会被渲染在这个位置。

接下来，可以看到一个带有if属性的tpl行为修饰符❸，它执行this.isCamry并传入values。this.isCamry方法是在XTemplate末尾处生成的❺。我们稍微再来说它。if属性更像是一个if条件，如果条件满足XTemplate就将生成HTML片段。在这里，this.isCamry必须返回true，封装在这个tpl标记中的HTML片段才可以生成。

values属性是对要循环访问的数组的值的一个内部引用。因为要循环访问一个字符串数组，它引用单个字符串，也就是一辆车的名字。

在下一行中，强制执行JavaScript代码❹。任何被括在花括号和方括号里的内容（{[...JScode...]}）都将被解读为通用JavaScript；它可以访问XTemplate提供的一些本地变量，还可以随着循环的每一次重复而改变。这样，检查当天索引（xindex）是否小于数组（xcount）里的元素个数，并返回一个逗号带一个空格，或者返回一个空字符串。执行这个检查用的内联函数可以确保逗号被放在车的名字之间。

最后一个值得注意的地方是，容纳isCamry方法的对象❺。导入一个有一套成员并向XTemplate构造函数传入参数的对象（或者对对象的引用），将导致那些成员被直接应用于XTemplate本身的实例。这就是为什么直接在其中一个tpl行为修饰符伪元素的if条件中直接调用this.isCamry。所有这些成员方法都是在它们被传入的XTemplate实例的作用域范围内调用的。这个概念非常强大但也可能有风险，因为可能会覆盖一个现有的XTemplate成员。所以请设法让方法或者属性具唯一性。isCamry方法使用JavaScript简写检查传入的字符串，也就是car变量，是否是"Camry"，如果是会返回true，否则会返回false。图2-9展现了XTemplate高阶示例的运行结果。

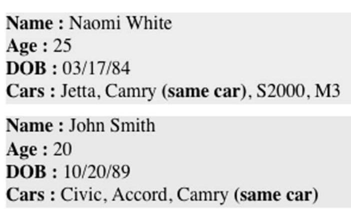


图2-9　XTemplate高阶示例的运行结果

结果显示所有行为注入的运行效果都符合预期。所有车都被列出，逗号的位置都正确。可以看出强制JavaScript注入奏效了，因为字符串"(same car)"显示在了Camry名字的右侧。

如你所见，相比使用Ext.Element配合数据生成HTML片段的通用DOM注入方法，模板和XTemplate有很多好处。我们鼓励你多看看模板和XTemplate的API页面，了解更多关于如何使用这些工具的细节和示例。你下一次接触模板将是在你学习如何创建自定义ComboBox（组合框）的时候。

2.4　小结

本章中，我们讨论了JavaScript应用逻辑在过去是如何通过<body>元素的onLoad处理程序加载的。要记住通常来说不同的浏览器，在DOM准备好进行操作的时候有各自不同的发布方式，这会导致代码管理很棘手。在学习使用Ext.onReady的过程中，你了解到它负责让不同的浏览器在恰当的时候启动你的程序，让你可以集中精力处理重要的东西：应用逻辑。

然后你深入了解了Ext.Element类，它封装了DOM节点，并为DOM节点提供了端对端管理。你通过对元素的增加和删除，探索了几个针对DOM节点的管理工具。所有UI部件都使用Ext.Element，这也使之成为使用最多的核心框架组件。每个部件的元素都可以通过公共的getEl方法，或者私有的el属性加以访问，但前提是它已经被渲染过。

最后，你了解了如何使用Template类往DOM中注入HTML片段，还体验了XTemplate的高阶用法，学习了如何往模块定义本身嵌入行为修饰逻辑，并根据提供的数据产生出结果。

在后面，你将专注学习框架的UI，并深入探索驱动框架的核心理念和模型。

第 3 章

组件和容器

本章内容

- 了解组件模型和组件生命周期
- 探索Ext JS容器模型
- 管理部件的父子关系
- 实现容器模型工具方法

我（Jesus）还记得最初使用Ext框架的时候通过示例和阅读API文档来进行学习。当时，我花了很多时间来学习一些最重要的核心UI概念，包括添加用户交互、重用部件以及理解一个部件如何包含或控制另一个部件。比如说，如何实现当点击链接标签时显示一个Ext窗口的功能。当然，有一种JavaScript方法是增加一个事件处理程序，但我想使用Ext JS。同样，我需要知道如何让部件彼此实现沟通。比如说，如何实现在点击一个网格面板的一行时，重新加载另一个网格面板。另外，再比如如何动态地往面板上添加子元素以及动态地移除子元素，如何在一个基于类型输入框的表单面板内找到一个特定的输入框。

本章里，你将探索基础的UI构建块——`Component`类，并了解它是如何通过实现一个称为组件生命周期的标准行为模板，胜任所有UI部件的核心模型的。

我们还将讨论`Container`类，并详细介绍部件是如何管理子元素的。你将了解如何往面板等部件上动态添加和移除子元素，它们可以用作动态更新UI的构建块。

3.1 组件模型

Ext JS组件模型是一个集中式模型，提供了很多与组件相关的任务，包括一套称为“组件生命周期”的规则，它决定了组件如何实例化、渲染以及销毁。我们将在3.2节讨论组件生命周期。

所有UI部件都是`Ext.Component`的子类，这也就意味着所有部件都遵从该模型制定的规则。图3-1部分展示了哪些类型的部件是`Component`类的子类。

了解每一种UI部件会如何运行，可以使框架具有稳定性和可预测性。组件模型也支持类的直接实例化，也就是延迟实例化，即所谓的XTypes。知道什么时候该用什么可以增强应用的响应能力。本节将讨论组件模型的三种基本特性。

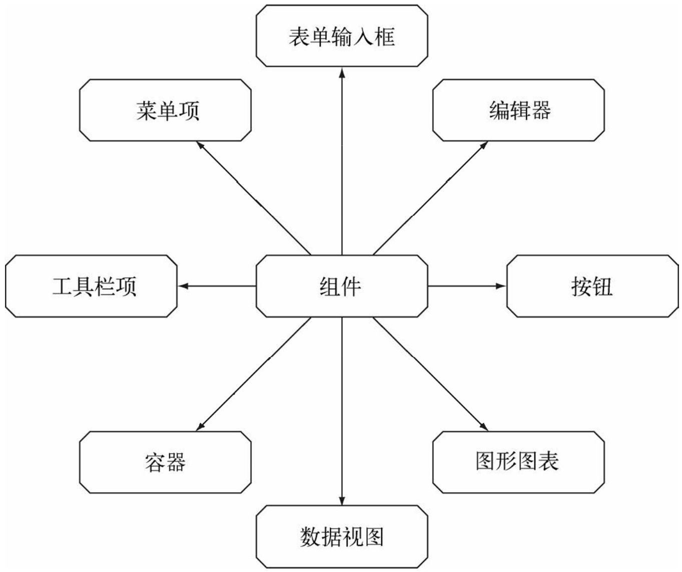


图3-1　Ext JS的Component类在该框架的所有UI部件中都扮演了重要角色

3.1.1　XType和ComponentManager

Ext 2.0引入了XType这一全新概念，它允许组件延迟实例化。XType可以加速复杂用户界面的类实例化，并大大提高代码的整洁程度。

简而言之，XType无非是一个纯JavaScript对象，它通常包含一个xtype属性，其中有一个字符串值标示该XType对应的类。以下是一个实际使用XType的简单示例：

```
var myPanel = {
    xtype  : 'panel',
    height : 100,
    width  : 100,
    html   : 'Hello!'
};
```

这里的myPanel是一个XType配置对象，将用于配置一个Ext.Panel部件。这之所以能行，是因为几乎所有部件都通过唯一的字符串值和一个对该类的引用，注册到Ext.ComponentMananger类，然后再被引用为一个XType。在该框架的每个UI类中，你都会找到一个以'widget.'为前缀的alias声明，Ext JS类系统会根据该前缀自动把部件的XType注册到ComponentMananger。

以下是为一个自定义类注册XType的代码：

```
Ext.define('MyApp.CustomClass', {
    extend: 'Ext.panel.Panel',
    alias: 'widget.myCustomComponent'
});
```

一旦注册完成，就可以把自定义组件指定为一个XType：

```
new Ext.Panel({
    ...
    items : {
        xtype : 'myCustomComponent',
        ...
    }
});
```


在初始化一个包含子元素的可视化组件时，它会检查自己是否有this.items，并检查this.items里是否有XType的配置对象。如果找到配置对象，它会尝试用ComponentManager.create创建一个该组件的实例。如果xtype属性在配置对象中没有定义，则可视化组件会在调用ComponentManager.create的时候定义它的defaultType属性。

这个概念乍听上去可能有点令人困惑。为了更好地理解它，要用accordion布局（折叠布局）来创建一个包含两个子元素的窗口，其中之一不包含xtype属性。首先，为两个子元素创建配置对象：

```
var panel1 = {
    xtype : 'panel',
    title : 'Plain Panel',
    html  : 'Panel with an xtype specified'
};
var panel2 = {
    title : 'Plain Panel 2',
    html  : 'Panel with <b>no</b> xtype specified'
};
```

请注意，panel1有一个显式xtype值'panel'，之后它将被用于创建一个Ext.Panel的实例。对象panel1和panel2很相似，但它们有一个明显的区别：对象panel1指定了一个xtype，而panel2没有。

接下来创建窗口，它要用到这些xtype：

```
Ext.create('Ext.window.Window',{
    width: 200,
    height : 150,
    title  : 'Accordion window',
    border :  false,
    layout : {
        type    : 'accordion',
        animate : true
    },
    items : [
        panel1,
        panel2
    ]
}).show();
```

在实例化Ext JS窗口时，传入items，后者是一个引用数组，包含了对之前创建的配置对象的引用。渲染好的窗口应该像图3-2这样。点击一个折叠面板将使它展开，并折叠所有其他展开的面板，而点击展开的面板将使它折叠。

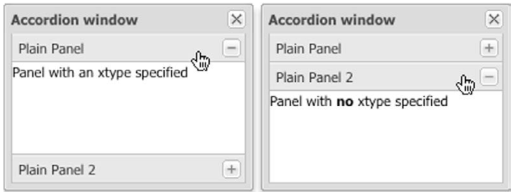


图3-2　XType示例的运行结果：一个Ext JS窗口，其中有两个根据XType配置对象生成的子面板

使用XType的一个鲜为人知的好处是，可以使写出的代码更简洁一些。因为可以用纯对象表示法，所以以内联方式指定所有XType子项，代码会更精简。以下是对前面案例的修改，包含了其所有子元素的内联：

```
Ext.create('Ext.window.Window',{
    width        : 200,
    height       : 150,
    title        : 'Accordion window',
    layout       : 'accordion',
    border       :  false,
    layoutConfig : {
        animate : true
    },
    items : [
        {
            xtype : 'panel',
            title : 'Plain Panel',
            html  : 'Panel with an xtype specified'
        },
        {
            title : 'Plain Panel 2',
            html  : 'Panel with <b>no</b> xtype specified'
        }
    ]
}).show();
```

可以看到，这段代码用`Window`配置对象以内联方式包含了所有子配置项。通过如此简单的示例无法看到使用XType所带来的性能提升。XType所带来的最大程度的性能提升体现在那些大型应用中，其中有相当多的组件需要实例化。

组件还包含另一个提升性能的特性：懒惰渲染。懒惰渲染意味着一个组件只有在必要的时候才被渲染。

3.1.2　组件渲染

`Ext.Component`类支持直接渲染和懒惰（按需）渲染这两种模式。直接渲染可以在`Component`的一个子类通过`renderTo`或`applyTo`属性实例化时进行，`renderTo`指向该组件借以渲染自身的一个引用，而`applyTo`则是引用一个元素，该元素的HTML结构令组件可以根据被

引用的HTML创建自己的子元素。通常是希望一个组件在实例化的同时被渲染，才使用这些参数，如以下示例所示：

```
var myPanel = Ext.create('Ext.panel.Panel',{
    renderTo : document.body,
    height   : 50,
    width    : 150,
    title    : 'Panel rendered immediately',
    frame    : true
});
```

这段代码的运行结果是Ext.panel.Panel得到即刻渲染，有些时候需要这样，有些时候则不需要。不需要即刻渲染的情况是，希望把渲染推迟到代码执行的另一时间，或者该组件是另一个组件的子组件。

如果想推迟组件的渲染，那就忽略renderTo和applyTo属性，并在觉得必要的时候调用该组件的render方法：

```
var myPanel = Ext.create('Ext.panel.Panel',{
    height : 50,
    width  : 150,
    title  : 'Lazy rendered Panel',
    frame  : true
});
// ……业务逻辑代码……
myPanel.render(document.body);
```

在这个示例中，创建了Ext.panel.Panel的一个实例，并创建了一个对它的引用myPanel。在一些假设的应用逻辑后，调用myPanel.render，并将一个引用传入document. body，可以把面板渲染到文档主体。

也可以向render方法中传入某元素的一个ID：

```
myPanel.render('someDivId');
```

在往render方法中传入一个元素ID的时候，组件将使用那个ID配合Ext.get来管理该元素，而元素被存储在它的本地el属性中。如果你有似曾相识的感觉，可能还记得上一章里关于Ext.Element的讨论，其中通过访问一个部件的el属性或者使用它的*访问方法*（accessor method）getEL来实现对它的引用。

这一规则有一个重大例外，那就是当该组件是另一个组件的子组件时，*绝不能*指定applyTo或者renderTo。包含其他组件的组件有一个父子关系，也称为*容器模型*。如果一个组件是另一个组件的子组件，它就会在配置对象的items属性中指定，而它的父组件将在必要的时候管理对其render方法的调用。这称为*懒惰渲染*或者*延时渲染*。

我们将在本章后面的内容中探讨容器，你将了解更多关于组件间父子关系的内容，但首先必须理解组件的生命周期，它详细规定了组件是如何创建、渲染并最终销毁的。了解每个生命周期各个阶段的运行原理，将让你更有能力打造健壮和动态的用户界面，并且有助于排查解决问题。

3.2　组件生命周期

与现实世界中的万物一样，Ext JS组件也有一个被创造、使用和销毁的生命周期。这个生命周期被划分为三个主要阶段：初始化、渲染和销毁、如图3-3所示。

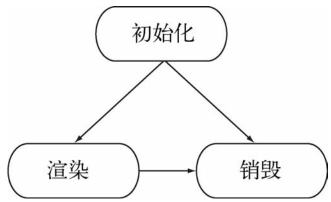


图3-3　Ext JS组件生命周期总是始于初始化，也总是以销毁结束。组件并非要进入渲染阶段才可以被销毁

为了更好地利用这个框架，你必须深入细致地了解这个生命周期的运行原理。如果要开发扩展、插件或者合成组件，这一点尤为重要。组件生命周期的每一阶段都要经历好几步，这是通过基类`Ext.Component`来控制的。

3.2.1　初始化

初始化阶段就是一个组件诞生的阶段。所有必需的配置设定、事件注册和预渲染过程都发生在这个阶段，如图3-4所示。

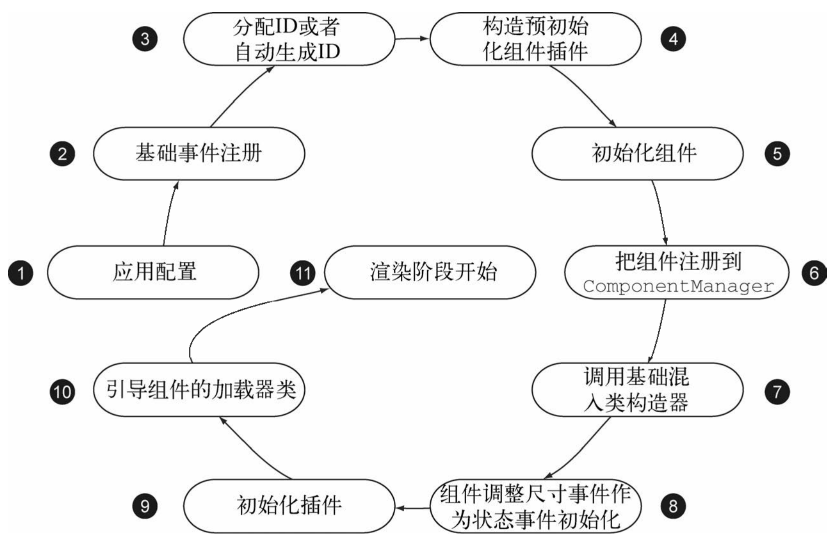


图3-4　在组件生命周期的初始化阶段会执行一些重要步骤，比如事件和组件注册，以及调用`initComponent`方法。很重要的一点是要记住，一个组件可以被实例化但不一定被渲染

让我们来探讨一下初始化阶段的各步操作。

(1) 应用配置。在实例化一个组件的时候，传入一个配置对象，配置对象包含了组件实现其功能所必需的全部参数和引用。这一步是在Ext.Component部件基类的开头几行完成的。

(2) 注册基础组件事件。根据组件模型，Ext.Component的每个子类都默认拥有一套由基类触发的核心事件。这些事件在以下一些行为发生前后触发：启用/禁用、显示、隐藏、渲染、销毁、状态还原和状态保存。上一个事件触发后，可以测试是否能够成功返回一个被注册的事件处理程序，并在执行任何实际的操作之前取消该行为。例如，当调用myPanel.show时，它会触发beforeshow事件，后者将执行为该事件注册的任何方法。如果beforeshow事件处理程序返回false，myPanel就不会显示。

(3) 分配或者自动生成组件ID。如果没有为组件配置静态ID，它会合并该组件的XType和一个从1000开始自动生成的数值，自动生成一个ID。比如说，创建一个Ext.panel.Panel的实例而不给它配置静态ID，结果该组件就会有一个类似于“panel-1001”的自动生成的ID。

(4) 构造预初始化组件插件。执行initComponent方法之前定义的任何插件，都是在这一步构造的。这可以让网格面板编辑器之类的插件在很早的时候就执行某些操作，比如初始化编辑器输入框。

(5) 运行initComponent。Component的子类在initComponent方法中执行大量操作，比如注册子类特有事件、引用数据存储以及创建子组件。initComponent被用作对构造器的补充，而且通常被用作扩展Component或者其子类的主要切入点。稍后我们将详细介绍如何用initComponent实现扩展。

(6) 注册ComponentManager。每一个被实例化的组件都通过一个Ext JS生成的唯一字符串ID，被注册到ComponentManager类。你可以选择在传入构造器的配置对象中传入一个id参数来覆盖Ext JS生成的ID。要当心的是，如果用一个非唯一的注册ID发出一个注册请求，则最新的注册将覆盖之前的注册。如果你打算使用自己的ID，一定要确保使用唯一的ID。

(7) 调用基础混入类构造器。组件利用两个混入类来提供低级别功能。Ext.util.Observable赋予了组件侦听和触发事件的能力，而Ext.state.Stateful则负责为组件处理状态特有事件。

(8) 启动调整尺寸状态事件。在这一步Component把它的基础调整尺寸事件注册为一个状态特有事件。这意味着Component的任何子类都有可能感知状态以调整尺寸。

(9) 初始化插件。如果在配置对象中把插件传入构造器，插件的Init方法将被调用，而其父Component将作为引用传入。很重要的一点是要记住，插件是根据它们被引用的顺序调用的。

(10) 引导组件的加载器。如果组件是用一个loader配置属性配置的，那它就被用来构造一个ComponentLoader的实例。这个类负责通过Ajax为一个组件获取数据，并利用一段HTML、一个数据或者一个组件渲染器来显示数据。详细信息参见Ext.ComponentLoader文档。

(11) 渲染组件。如果renderTo或者applyTo参数被传入构造器，那渲染阶段就开始了；否则，组件会保持休眠，等待代码或者它的一个父组件调用它的render方法。

组件生命周期这个阶段通常是最快的，因为所有工作都是在JavaScript中完成的。特别重要的一点是要记住，组件并不一定要渲染过才能销毁。

3.2.2　渲染

在渲染阶段，你可以获得组件已经成功初始化的可视化反馈信息。如果初始化因为某种原因失败了，那么该组件可能不会正确渲染或者根本不会渲染。对复杂的组件来说，这时会消耗掉很多CPU周期，会要求浏览器绘制屏幕，会进行运算以确保该组件的所有元素都可以获得正确的布局和尺寸。图3-5展现了渲染阶段的步骤。

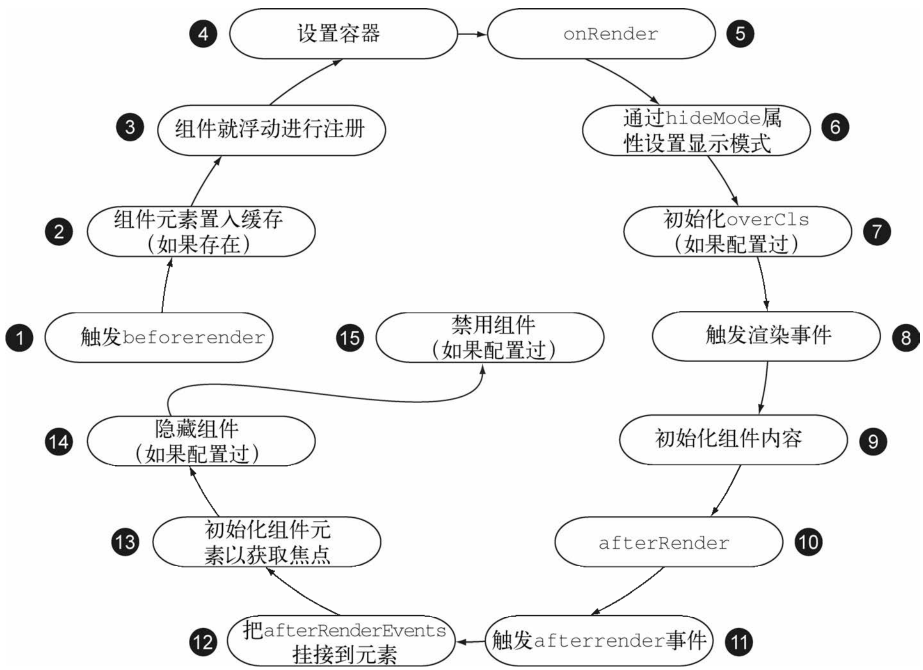


图3-5　一个组件生命周期的渲染阶段可能会占用大量CPU，因为它需要把元素添加到DOM，还需要进行运算以恰当地按尺寸生成并管理元素

如果没有指定`renderTo`或者`applyTo`，则必须要调用一次`render`方法，触发该阶段。如果组件不是另一个Ext JS组件的子组件，代码必须要调用`render`方法，并传入DOM元素的一个引用：

```
someComponent.render('someDivId');
```

如果该组件是另一个组件的子组件，它的`render`方法将会被其父组件调用。让我们来探索一下渲染阶段的各个步骤。

(1) 触发`beforerender`。组件触发`beforerender`事件，并且检查任何已注册的事件处理程序的返回值。如果一个已注册的事件处理程序返回`false`，组件就中止渲染行为。回想一下，初始化阶段的第二步为`Component`的子类注册了核心事件，而“before”事件可以中止执行行为。

(2) 组件的元素被置入缓存。如果配置了一个带el属性的组件，那它会围绕该元素封装一个xt.Element的实例。

(3) 组件注册为可浮动。如果组件配置为可浮动，它将把自己注册到WindowManager，以启用z-index和焦点管理。这一步对于Window和Menu等类很重要，它们被设计为定位（浮动）于其他UI部件上方。

(4) 设置容器。组件需要有一个容身之处，而这个容身之处就是它的容器。如果指定一个对某元素的renderTo引用，组件就往被引用的元素（也就是它的容器）上添加一个子div元素，并且在新添加的子元素内部渲染该组件。如果指定了一个applyTo元素，在applyTo参数中指定的被引用元素就成了组件的容器，而组件只把那些渲染它所需要的元素添加到被引用的元素上。然后该组件对在applyTo中引用的DOM元素就有了完全的管理权。当组件是另一个组件的子组件时，通常这两者都不会传入，在这里容器也就是父组件。有一点要注意，那就是你应该只传入renderTo或者applyTo，而不是两者同时传入。我们将在稍后学习更多关于部件的知识时探索renderTo和applyTo。

(5) 执行onRender。对Component的子类来说这是至关重要的一步，所有DOM元素都在这一步插入，以渲染组件并将其绘制在屏幕上。在之后扩展Ext.Component或者任何子类的时候，每个子类原则上都应该首先调用它超类的onRender，而这确保Ext.Component基类可以插入渲染一个组件所需的核心DOM元素。

(6) 配置显示模式。组件的元素被指示根据组件hideMode属性的设置情况，设置其显示模式。通常来说，不用担心应该把hideMode设置成什么，不过了解可用的选项总是好事，以防你要根据特殊的HTML需求创建一个自定义组件。它默认被设置为'display'（CSS display : none;），但其他的可选项有'visibility'（CSS visibility:hidden;）和'offsets'。hideMode设置成'offsets'可以让组件的元素就位，并在*X*和*Y*坐标轴上负偏移1000像素。

(7) 初始化overCls（如果配置过）。如果给一个组件配置overCls属性，则组件会指示其元素在它的mouseover事件上添加该CSS类，并在mouseout事件上设置removeOverCLs属性。

(8) 触发render事件。在这一步所有必需的元素都已经被插入DOM，并且都应用了样式。触发render事件，触发任何已注册的事件监听器。

(9) 初始化组件的内容。如果组件配置了contentEl、html和/或tpl（Template）和data属性的组合，则它会把相应的内容作为自己元素的子元素渲染。AbstractComponent经过模型化设计，如果你愿意的话可以使用一个、两个或者全部（三个）属性。html将首先渲染，然后是contentEl，接着是tpl，最后是data。

(10) 执行afterRender。afterRender是一个至关重要的渲染后方法，由render方法自动调用。这个方法负责配置组件的尺寸，排列和定位组件，并往HTML内容添加样式。它也负责初始化Resizable的一个实例（如果配置过），把组件的元素设置为可滚动（如果通过autoScroll配置过），并使组件可拖动（如果做过相应配置）。最后，如果组件就兼容富网络应用（ARIA）做过配置，则部件也进行相应的初始化。有一点要注意，那就是所有带afterRender方法的Component子类都应该要调用其超类的afterRender方法。

(11) 触发afterrender事件。这个事件对于类的关键之处在于，让类知道所有关键渲染操作都已完毕，组件已经做好修改其元素的准备。为让你的子类可以执行DOM操作，这个事件通常是最佳监听对象。

(12) 挂接afterRenderEvents。如果组件设置了一个afterRenderEvents配置对象，它将使用该配置对象把元素监听器挂接到可挂接的部件特有的元素引用，比如Component的el和Panel的body。

(13) 初始化组件的元素以获取焦点。被配置为可聚焦的组件和有可聚焦元素的组件将绑定到内部的onFocus处理程序上。这个必不可少的底层命令负责处理那些在管理了底层元素的focus和blur事件后，可以触发自身自定义focus和blue事件的组件。

(14) 隐藏组件。如果组件的配置中hidden设为true，则执行元素的hide方法。这样可以通过配置的hideMode属性来隐藏该组件。

(15) 禁用组件。如果组件的配置中disabled设为true，则执行组件的disable方法，这样可以有效地禁用该部件。有一点要注意，那就是禁用部件没有触发disable方法。

通常来说，组件生命周期的大部分都处于渲染阶段，直到它在销毁阶段消亡。

3.2.3 销毁

和现实中的生命一样，组件的消亡也是其生命周期中的关键阶段。销毁一个组件要执行一些关键任务，比如把它自身和所有子组件从DOM树中移除，并撤销该组件在ComponentManager类的注册，以及撤销事件监听器的注册，如图3-6所示。

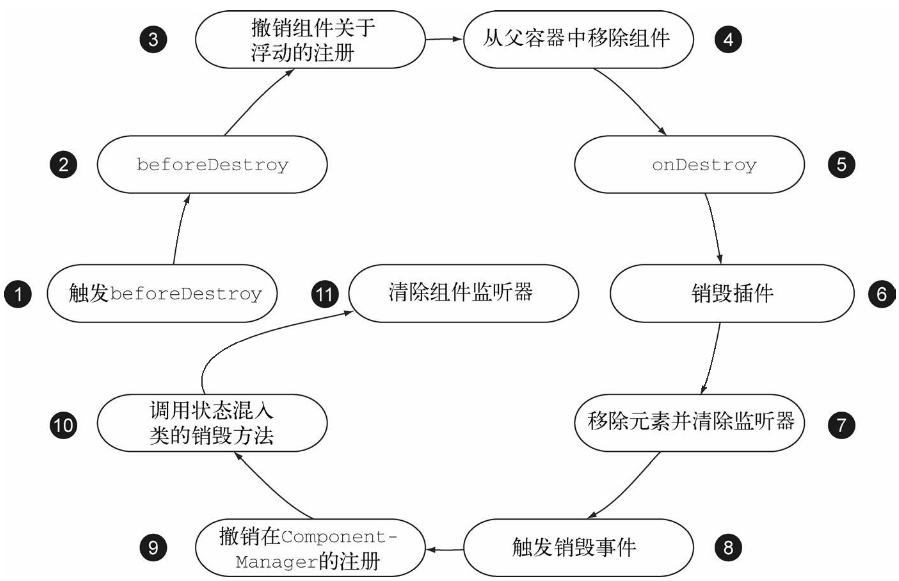


图3-6 组件生命周期的销毁阶段跟它的初始化阶段同等重要，因为事件监听器和DOM元素必须要撤销注册并移除，以减少总体的内存占用

组件的destroy方法可以由其父容器或者由代码实施调用。以下是组件生命周期最后这一阶段的步骤。

(1) 触发beforedestroy。这个事件跟很多before<action>事件一样，是可取消事件，以防组件在事件处理程序返回false的时候被销毁。

(2) 调用beforeDestroy。这个方法是组件的destroy方法中最早调用的一个，也是移除任何非组件元素（比如工具栏或者按钮）的绝佳时机。Component的任何子类都应该调用其超类的beforeDestroy方法。

(3) 撤销组件关于浮动的注册。如果一个组件是浮动的，它在FloatingManager的注册就被撤销。

(4) 从父容器中移除组件。如果组件是一个父容器的子组件，它就从父容器被移除。

(5) 调用onDestroy。onDestroy方法负责以下几项任务。首先是立刻销毁任何配置过的拖放代理。然后是销毁Resizer（如果配置过）。接下来，如果注册过焦点DelayedTask，就把它从组件中移除。如果组件被配置为监控浏览器的调整尺寸事件，则移除该调整尺寸事件处理程序。最后，如果组件配置了ComponentLayout、LoadMask和任何浮动子元素，则进行销毁。

(6) 销毁注册的插件。在销毁阶段的这一步，循环访问所有注册过的插件，调用它们各自的destroy方法。

(7) 清除元素和元素监听器。如果组件已经被渲染，则移除任何注册到其Element的处理程序，并将Element从DOM中移除。

(8) 触发destroy事件。这个事件触发任何注册过的事件处理程序，标志着该组件不在存在于DOM中。

(9) 撤销组件在ComponentManager的注册。移除ComponentManager类中对这个组件的引用。

(10) 销毁状态混入类。在这一步调用状态混入类以进行销毁，撤销任何状态特有的组件事件的注册。

(11) 清除组件的事件处理程序。撤销所有事件处理程序在组件的注册。

以上就是对组件生命周期的深入探究，而组件生命周期也正是令Ext JS框架如此强大和成功的特性之一。

如果你打算开发自定义组件，请不要忽略组件生命周期的销毁环节。很多开发者就是因为忽略了这至关重要的一步而陷入麻烦，在代码中残留了工件，比如持续轮询Web服务器的绑定数据存储，或者是预设DOM中有元素存在的事件监听器。如果不妥善地清理掉它们，可能会导致异常，并中止执行一个至关重要的逻辑分支。

接下来，我们要来看Container类，它是Component的一个子类，赋予了组件管理父子关系中其他组件的能力。

3.3 容器

Container是一个幕后类，它给组件管理其子元素提供了基础，而它经常被开发者忽视。这

个类提供了一整套工具方法，其中包括add、insert和remove方法，还有query、bubble和cascade等子工具方法。这些方法被大多数子类使用，包括Panel、Viewport和Window。在你的应用中使用这些类也同样很常见。

3.3.1　构建一个带子元素的容器

为了学习这些工具的原理，你需要构建一个包含一些子元素的容器待用，如代码清单3-1所示。

代码清单3-1　创建第一个容器

```
var panel1 = {                                     ❶ 配置面板
    html   : 'I am Panel1',
    id     : 'panel1',
    frame  : true,
    height : 100
};
var panel2 = {
    html  : '<b>I am Panel2</b>',
    id     : 'panel2',
    frame : true
};

var myWin = Ext.create('Ext.window.Window',{         ❷ 创建窗口
    id     : 'myWin',
    height : 400,
    width  : 400,
    items  : [
        panel1,
        panel2
    ]
});
myWin.show();
```

我们看一下代码清单3-1。你要做的第一件事，是创建两个普通的面板❶，然后创建myWin❷，它是Ext.window.Window的一个实例，包含之前定义好的面板。渲染过的UI应该如图3-7那样。

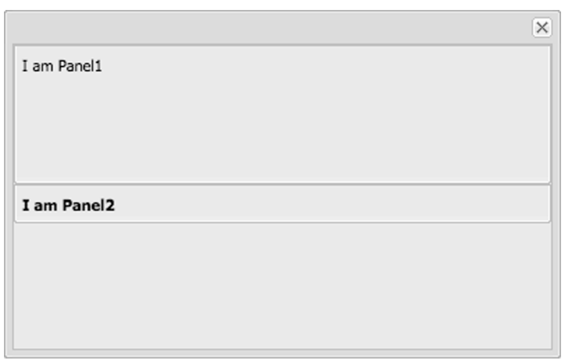


图3-7　利用代码清单3-1渲染后的容器UI

你要在myWin的底部留出一些空间，在添加元素的时候会用得上。每一个容器都通过一个items属性保存对其子元素的引用，该属性是Ext.util.MixedCollection的一个实例，可以通过someContainer.items访问。

MixedCollection是一个工具类，它可以让框架保存一个混合数据集（包括字符串、数组和对象），并为其建立索引。它还提供了一系列便利的工具方法。

现在你已经渲染好了容器，让我们来给它添加子元素。

3.3.2 处理子元素

和在现实世界中应对子女一样，处理子元素也会让你纠结地生出白发，所以必须要学习使用可用的那些工具。掌握了这些工具方法，你就可以动态更新UI，这正是Ajax网页的精髓。

添加组件是一项简单的任务，有两个方法可以选择：add和insert。add方法只会往容器的层级上添加一个子元素，但使用insert可以在一个特定的索引位置往容器中插入一个元素。

我们来往代码清单3-1中创建的容器上添加元素，为此要用到便捷的Firebug JavaScript控制台：

```
Ext.getCmp('myWin').add({
    title : 'Appended Panel',
    id    : 'addedPanel',
    html  : 'Hello there!'
});
```

运行这段代码将把元素添加到容器，参见图3-8。

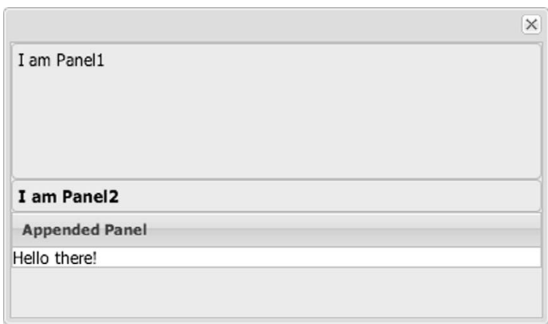


图3-8 往代码清单3-1中的窗口添加一个面板

添加子元素很容易，但有时候必须能够把元素插入到特定的索引位置。使用insert方法可以轻松完成这项任务：

```
Ext.getCmp('myWin').insert(1, {
    title : 'Inserted Panel',
    id    : 'insertedPanel',
    html  : 'It is cool here!'
});
```

在索引位置1插入一个新面板，它就在Panel1下方。结果应该如图3-9所示。

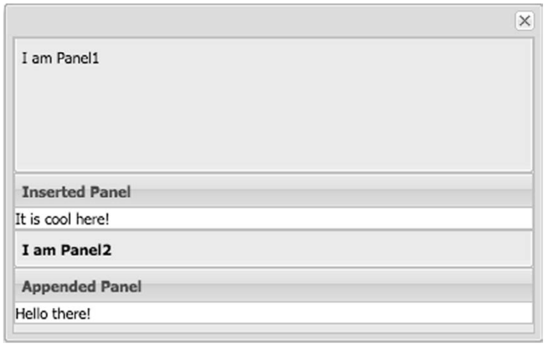


图3-9　动态添加和插入的子面板渲染后的结果

可以看到，添加和插入子组件毫不费力。移除元素也同样容易，这一操作需要两个参数。第一个参数是对想要移除子元素的组件或者组件ID的引用。而第二个参数指定应不应该为该组件调用destroy方法，当想把组件从一个容器转移到另一个容器的时候，这一参数将赋予你极强的灵活性。在便捷的Firebug控制台上，要像下面这样移除最近添加的一个子面板：

```
var panel = Ext.getCmp('addedPanel');
Ext.getCmp('myWin').remove(panel);
```

执行这段代码后，会发现面板立刻就消失了，这是因为没有指定第二个参数，它的默认值是true。可以通过把一个父容器的autoDestroy设置为false，来覆盖这个默认参数。当移除一个组件时，调用该组件的destroy方法，启动它的销毁阶段并删除其DOM元素。

如果想把一个子元素移动到一个不同的容器中，就要把remove方法的第二个参数指定为false，然后往父容器中添加或插入该子元素，如下所示：

```
var panel = Ext.getCmp('insertedPanel');
Ext.getCmp('myWin').remove(panel, false);
Ext.getCmp('otherParent').add(panel);
```

上面的这段代码假定已经有另一个以'otherParent'的ID实例化的父容器。创建一个对之前插入面板的引用，并从它的父容器中将其非破坏性移除。接下来，把它添加到它的新父容器中，实施DOM级的移动操作，把子组件的元素移动到新父容器的内容体元素中。

Container类提供的工具方法不只是对子元素的添加和移除操作，还让你能够深入该容器的层级以寻找子组件。如果想搜集一个特定类型或者满足特殊标准的子元素的列表，并对它们实施一项操作，这就会很有用。

3.4　查询组件

Ext JS 4.0带有一个新的ComponentQuery类，它有一个选择器引擎，类似于浏览器的选择器

引擎，这意味着在搜寻组件的时候可以使用熟悉的查询构造语句。因为`ComponentQuery`是模仿浏览器的选择器引擎设计的，所以可以以任何源节点为根执行查找操作。

为了加以演示，我们创建了一个包含深层嵌套面板的示例。要查看这一工具，请访问以下地址：http://extjsinaction.com/v4/examples/ch03/using_ComponentQuery.html。在图3-10中，我们要使用Firefox的Firebug插件的多行控制台模式。

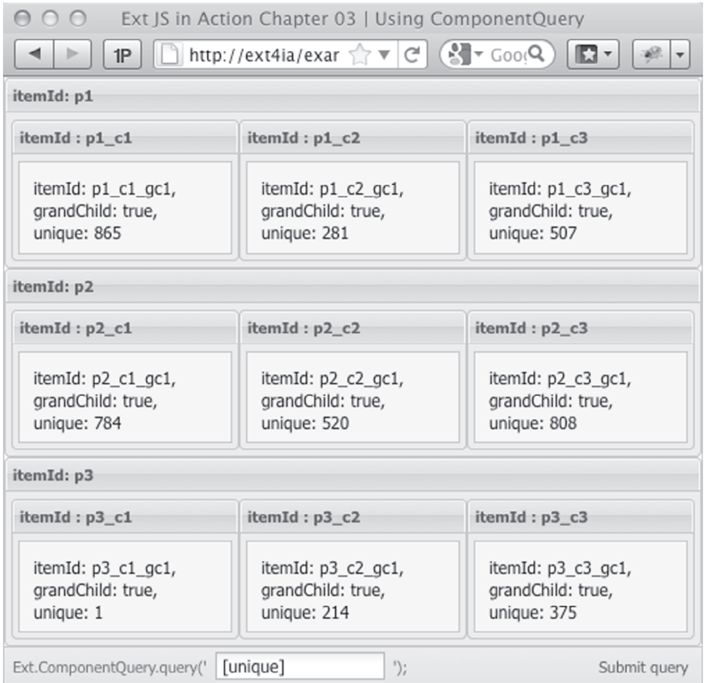


图3-10 将在`ComponentQuery`示例中用到的工具

这个在线工具包含了多个四层嵌套的子组件，首先是一个使用Fit布局、渲染了单个“主”面板的视口。这个主面板有一个`master_panel`的`itemId`，并包含三个子面板。往前移的子面板的`itemId`显示在它们的标题或者内容体中。嵌套最深的组件有一个`grandchild`属性，被设为布尔值`true`。你将看到，这些已知的属性都可以用于追踪任何`targetComponent`。最后，每个子组件都有一个名为`unique`的唯一属性，它是随机生成的，并随着每一次页面刷新而改变。

首先要取得最顶端面板的一个引用。在这里和示例的其他部分，都要使用Firebug控制台，所以在继续阅读后续内容的时候请保持它开启可用。

注意 如果没有Firebug，我们已经导入了一个工具栏，可以帮你实施查询。你唯一要做的就是提供查询参数，然后点击“提交”按钮。找到的任何组件都将淡出并再次淡入以表明它们被发现了。

在Firebug的JavaScript控制台中输入并且执行以下代码。你会发现整个应用淡出和淡入：

```
var results = Ext.ComponentQuery.query('#master_panel'),
    panel   = results[0];
panel.body.fadeOut().fadeIn();
```

在这段代码中首先创建一个results引用，指向查询调用的结果。配置的查询将搜寻并返回任何拥有master_panel的id或者itemId的已注册组件。

然后设置panel引用，指向返回的结果集的第一个元素。不管找到了多少元素，ComponentQuery的query方法都会默认返回一个数组。这就意味着要测试查询是否成功，必须要检查返回的结果集中的结果数，看是不是大于0。

如果打算寻找唯一的结果，也可以把前两行代码缩减为这样的一行：

```
var panel = Ext.ComponentQuery.query('#master_panel')[0];
```

刚刚实施的是全局性查询。Ext JS在页面上搜索所有注册的组件，这是相对高级别的查询。但也可以深入搜索任何有特定属性的组件。

比如说，可以使用这种全局性查询机制来查询一个深层嵌套面板：

```
var panel = Ext.ComponentQuery.query('#p2_c3')[0];
```

此外还有一些其他的模式。比如说，如果想搜索一个属性是否存在，就可以这么做。要搜索grandchild属性是否存在，就查询[grandchild]。如果想搜索唯一的属性，比如设置的随机unique属性，则可以查询[unique="784"]。尽管该属性的值其实是一个整数，还是给它打上了双引号，指示Ext JS测试值的类型。

容器能够通过down方法顺藤摸瓜进行深层查询。这么做可以使用ComponentQuery机制，迫使搜索的范围局限在容器自身和子组件内部。与down方法相对的是up方法。子组件可以使用ComponentQuery顺着父层级往上查询，但查询的范围从子组件开始，然后直接上升到最顶层的父容器。

现在你已经有了管理子项所必需的核心知识。让我们转移一下注意力，来探索一些常用的Container的子类，感受一下Ext JS UI的强大。你将看到如何借助Ext JS利用浏览器所有能用的阅读空间创建一个UI。

3.5　视口容器

Viewport类是所有单纯依靠Ext JS开发的Web应用的基础。它管理着浏览器100%的视口，也就是显示区域。这个类的代码算起来只有20行出头，极为精简高效。 因为它是Container类的直接子类，所以所有子组件管理和布局功能都可以为你所用。要使用视口，先来尝试一下以下示例代码：

```
Ext.create('Ext.container.Viewport', {
    layout : 'border',
```

```
        items   : [
            {
                height : 75,
                region : 'north',
                title  : 'Does Santa live here?'
            },
            {
                width  : 150,
                region : 'west',
                title  : 'The west region rules'
            },
            {
                region :'center',
                title  :'No, this region rules!'
            }
        ]
    });
```

代码中渲染过的视口用到了整个浏览器的视口，并显示了三个用border布局加以组织的面板。如果调整浏览器窗口的尺寸，会注意到中央的面板是自动调整尺寸的，这展现了视口是如何监听并响应浏览器窗口的`resize`事件的（见图3-11）。`Viewport`类为所有基于Ext JS并把该框架用作富网络应用完整网页UI解决方案的应用提供了基础。

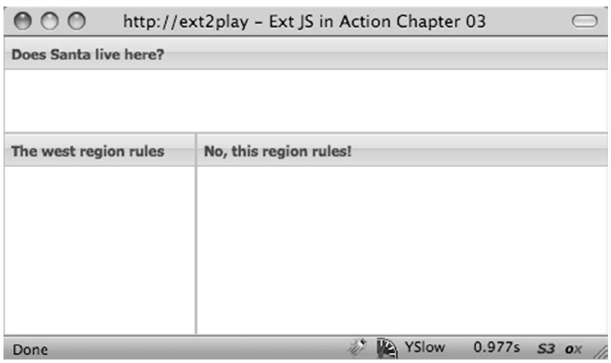


图3-11　第一个视口，它占据了浏览器全部的可用阅读空间

很多开发者试图在一个受到完全管理的Ext JS页面中，创建多个视口以显示多个界面，结果都碰了壁。要解决这个问题，你可以综合使用card布局和视口，在不同的应用程序界面之间切换，这些界面的大小经过调整以适应视口。我们将在第5章深入探讨布局，届时你将学到诸如fit和flip这样的关键词在布局上下文中是何含义。

3.6　小结

本章深入探索了组件模型，它赋予了Ext JS框架一个管理组件实例的统一方法。组件生命周期是该框架UI部分最重要的概念之一，这就是我们在深入学习众多部件之前就介绍它的原因。

之后你探索了容器的世界，学习了如何使用容器来管理子组件，了解了Viewport类，以及为何说它是完全基于Ext JS开发的Web应用的基础。

打好的这些基础对于学习使用Ext JS UI部件非常有帮助。接下来，我们要探讨面板，它是最常用的显示内容的UI部件。

Part 2

第二部分

Ext JS 组件

到了这里，你应该已经做好了深入学习Ext JS框架中各种部件的准备了。在这一部分，你将有机会学习运用数据驱动的视图。

第4章介绍了显示用户交互内容的核心UI组件，比如面板、窗口、消息框、选项卡、工具栏和按钮，以及管理子元素的容器生命周期。第5章探究了用于界面组织的布局，比如Fit布局、Card布局和Border布局。第6章研究了表单和不同的输入框，比如单行文本框、多行文本框、数字输入框和组合框，外加校验以及相关的数据存储和数据视图。

第7章解读了用于保存数组、JSON和XML数据的数据存储，用于数据驱动视图（包括数据模型、代理、读取器和写入器）的高级用户交互，以及关联、校验、网格面板和编辑器插件。第8章讨论了以表格形式实现快速输入的网格面板，并使用诸如菜单、交互、编辑、分页和滚屏的功能呈现和操作大型数据集。第9章介绍了用于演示层级数据的树，内容涵盖远程数据加载、编辑节点数据以及自定义背景菜单。第10章涉及的是用于呈现数据表示的图形和图表，以及它们的图表类型、形状、主题和说明。

第11章阐述了借助数据存储、网格面板、树、表单和模板，用于服务器数据交换的直接远程方法调用。第12章讨论了如何使用拖放工作流，借助于鼠标手势在屏幕上移动元素，以及相关联的覆盖方法、拖放生命周期，还有网格面板和树形面板插件。

最后，你将在Ext JS部件的工作原理以及如何高效运用它们方面打下扎实的基础。

第 4 章

核心UI组件

本章内容

- 探索面板
- 实现面板内容区
- 显示一个`Ext.Window`
- 使用`Ext.MessageBox`
- 创建标签面板

开始用Ext JS体验尝试或者开发应用的时候，开发者经常会首先复制可下载的SDK中的示例。虽然这种方法有利于学习如何实现一个特定的布局，但在解释该布局的工作原理方面有所欠缺，而这会让你头疼不已。在本章，你将学习开发成功UI方案所赖以为基础的那些核心主题。

你还将深入学习面板的工作原理，并探究面板上可以显示内容和UI部件的区域；然后学习窗口和消息框，后者浮动在页面的所有其他内容上方。另外，你还将学习标签面板。

当学完本章内容时，你将能够管理容器和它们子元素完整的CRUD（创建、读取、更新和删除）生命周期，为以后开发应用打下基础。

4.1 面板

`Panel`是`Container`的子类，被视为Ext JS框架里的劳模，因为很多开发者都用它来展示UI部件。一个完全加载的面板被划分为6个显示内容的区域，如图4-1所示。你可能还记得`Panel`还是`Component`的子类，这也就意味着它遵从组件的生命周期。在后面的内容中，我们要用容器这个词来表述任何`Container`的子类，因为我们希望强调"`Panel`是`Container`的子类"这个概念。

面板的标题栏是个很忙的地方，它为最终用户提供了可视和交互内容。而在微软Windows系统中，可以在面板的左上角放置一个图标，给用户们提供信息，比如他们看到的是哪种面板。除了图标以外，还可以在面板上显示一个标题。

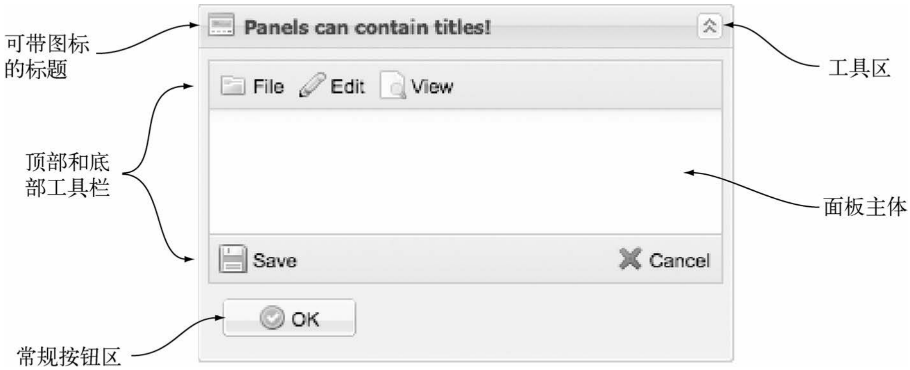


图4-1 一个完全加载的面板示例，它有一个带一个图标和多个工具的标题栏，有一个顶部和一个底部工具栏，底部还有一个按钮栏

在标题栏最右边的区域是一个工具区，这里显示了一些微型图标，点击它们会调用处理程序。Ext JS为工具提供了很多图标，其中包括很多与用户相关的常用功能，比如帮助、打印和保存。如果要显示所有可用的工具，就访问工具API的`type`属性。

在6个内容区中，面板主体大概是最重要的。它是主要内容或者子元素所在的地方。`Container`类规定，一个布局如果不想用容器的默认布局，就必须在实例化的时候指定布局。如果没有指定布局，那就要用到`AutoLayout`默认布局管理器。布局一个很重要的属性是，一个布局不能被动态转换为另一个布局。

我们来构建一个复杂的面板，它有顶部和底部工具栏，每个工具栏上各有两个按钮。

4.1.1 构建一个复杂的面板

因为工具栏有按钮，所以当按下按钮以显示反馈的时候，最好可以有个方法调用。这个方法称为处理程序：

```
var myBtnHandler = function(btn) {
  Ext.MessageBox.alert('You Clicked', btn.text);
}
```

当点击任何工具栏的按钮时，这个方法都将被调用。工具栏按钮会调用处理程序，并把它们自身作为一个引用`btn`传入处理程序。然后，定义工具栏，如代码清单4-1所示。

代码清单4-1 构建工具栏以备在面板中使用

```
var myBtnHandler = function(btn) {                       ◁─┐
        Ext.MessageBox.alert('You Clicked', btn.text);    ❶ 添加点击
    },                                                      处理程序
    fileBtn = Ext.create('Ext.button.Button', {          ◁─┐
        text    :'File',                                 ❷ 添加文件按钮
        handler : myBtnHandler
    }),
    editBtn = Ext.create('Ext.button.Button', {
```

```
        text     : 'Edit',
        handler  : myBtnHandler
    }),
    tbFill = new Ext.toolbar.Fill();

var myTopToolbar = Ext.create('Ext.toolbar.Toolbar', {      <-- ③ 实例化顶部工具栏
    items : [
        fileBtn,
        tbFill,
        editBtn
    ]
});

var myBottomToolbar = [                                     <-- ④ 配置底部工具栏
    {
        text    :'Save',
        handler : myBtnHandler
    },
    '-',
    {
        Text    :'Cancel',
        handler : myBtnHandler
    },
    '->',
    '<b>Items open:1</b>'
];
```

代码清单4-1提供了两种方法来定义一个工具栏及其子组件。首先定义`myBtnHandler`❶。每个按钮的处理程序默认有两个参数：`Button`对象本身，以及用一个`Ext.Event`对象封装的浏览器事件。使用传入的`Button`引用（`btn`），并把该文本传入`Ext.MessageBox.alert`，以获得按钮被点击的可视确认信息。

接下来实例化文件❷和编辑按钮，以及“贪婪”的工具栏空白，后者会把它后面的所有工具栏元素往右推。把`myTopToolbar`分配到`Ext.Toolbar`的一个新实例❸，把之前创建的按钮和工具栏空白，引用为新工具栏的`items`数组中的元素。

对一个相对简单的工具栏而言，这很是大费周章。我们这么做一遍是为了让你“感受”用这种老办法编程有多“痛苦”，更深刻地体会到Ext JS快捷方式和XType能节省多少时间（和最终开发代码）。

`myBottomToolbar`❹引用是一个对象和字符串组成的简单数组，当它的父容器觉得必要的时候把该数组转化为恰当的对象。你会用这两个方法在两个工具栏上动态地添加或者删除元素。接下来要创建面板主体：

```
var myPanel = Ext.create('Ext.panel.Panel', {
    width       : 200,
    height      : 150,
    title       : 'Ext Panels rock!',
    collapsible : true,
    renderTo    : Ext.getBody(),
    tbar        : myTopToolbar,
```

```
    bbar        : myBottomToolbar,
    html        : 'My first Toolbar Panel!'
});
```

你以前创建过面板，这里的所有代码看起来都应该很熟悉，除了tbar和bbar这两个属性，它们引用了新创建的工具栏。另外，还有一个collapsible属性，当collapsible设置为true的时候，面板会在标题栏的右上方创建一个开关按钮。渲染过后，面板的外观应该是如图4-2所示。记住，点击任何一个工具栏按钮，都会导致跳出一个Ext.MessageBox显示按钮上的文本，反馈点击处理程序被调用的确认信息。

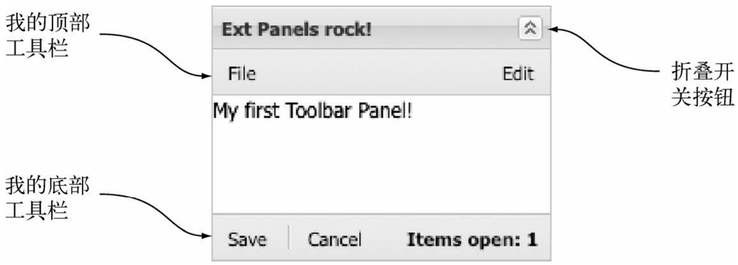


图4-2　代码清单4-1渲染后的结果，创建了一个复杂的可折叠面板，顶部和底部工具栏上分别都含有按钮

在图4-2中渲染的工具栏分别被置于内容体的上方和下方。这一过程称为停靠（docking），是由Dock组件布局类来管理的。我们将在4.1.2节深入探讨停靠。

4.1.2　添加按钮和工具

工具栏是放置面板主体以外的内容、按钮和菜单的好地方。有两个部分仍有待学习：按钮和工具。要在下面的代码清单中往myPanel示例添加按钮和工具，但要使用Ext JS快捷方式，以及内嵌了所有其他配置选项的XType。

代码清单4-2　向现存的面板上添加按钮和工具

```
var myPanel = Ext.create('Ext.panel.Panel', {
    // height、weight和renderTo属性
    buttonAlign : 'left',                          ❶ 添加按钮
    buttons     : [
        {
            text    : 'Press me!',
            handler : myBtnHandler
        }
    ],                                                 ❷ 配置工具
    Tools       :[
        {
            type    : 'gear',
            handler : function(evt, toolEl, panel) {
                var toolClassNames = toolEl.className.split(' ');
                var toolClass      = toolClassNames[1];
```

```
                var toolId         = toolClass.split('-')[2];

                Ext.MessageBox.alert('You Clicked', 'Tool ' + toolId);
            }
        },
        {
            Type    : 'help',
            handler : function() {
                Ext.MessageBox.alert('You Clicked', 'The help tool');
            }
        }
    ]
});
```

在代码清单4-2中，往之前的配置选项中引入了两个快捷方式数组：一个用于按钮，另一个用于工具。因为指定了一个buttons数组❶，在面板渲染的时候，它会创建一个新的带有一个特殊的CSS类 x-toolbar-footer-docked-bottom的Ext.Toolbar实例，然后把它渲染到新创建的页脚div上。“点我!”按钮会被渲染在新创建的页脚工具栏上，而当点击该按钮时，它会触发你之前定义的myBtnHandler方法。

如果观察代码清单4-1里的myBottomToolbar快捷方式数组和代码清单4-2里的buttons快捷方式数组，你会看到一些相似之处。所有的面板工具栏（tbar、bbar和buttons）都可以用相同的便捷语法定义，因为它们都将被转化为Ext.Toolbar的实例，并渲染到它们在面板中的相应位置。

你还指定了一个tools数组❷配置对象，这跟定义工具栏的方式有些不同。在这里要为工具设置图标，必须指定工具的id，比如'gear'或'help'。该数组中指定的每个工具中都会创建一个图标。Panel类将为每个工具分配一个点击事件处理程序，它将调用该工具的配置对象中指定的处理程序。新修改的myPanel渲染后的版本应该如图4-3所示。

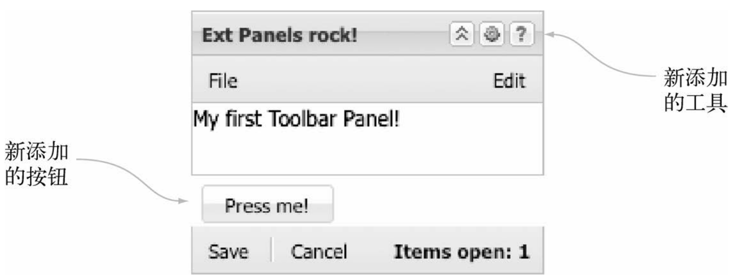


图4-3　代码清单4-2渲染后的结果，往按钮栏添加了一个按钮，往标题栏添加了工具

你可以看到为该面板配置的按钮，但有些不对劲。底部的工具栏被渲染在了按钮栏的下方，需要大修大补。这给了我们一个绝佳的机会，来深入探讨Panel类如何使用Dock组件布局渲染内容体上方、下方，甚至是左侧和右侧的元素。

4.1.3 在一个面板上停靠元素

Dock布局是Panel类特有的布局，作用是把一个或者多个组件渲染在面板内容体左右任意一侧。当在代码清单4-2中为面板配置tbar和bbar属性的时候，面板实质上就是在内部把配置推送入所谓的dockedItems。Dock布局的作用是组织排列每一个基于weight属性的部件。稍后我们再来探讨weight属性，首先我们要让你看到组件停靠的灵活性有多大。

不用为一个面板配置tbar和bbar（或者lbar和rbar）属性，而是可以把dockedItems配置为一个工具栏配置选项的数组。这么做可以把所有停靠元素保存在单个配置集合中。

接下来这个代码清单展示了如何不用tbar而用dockedItems，把单个顶部停靠工具栏停靠在面板上。我们会尽可能地让代码保持简单。

代码清单4-3 顶部停靠单个工具栏

```
var buttons = [                                          ❶ 添加按钮
    { text : 'Btn 1' },
    { text : 'Btn 2' },
    { text : 'Btn 3' }
];

var topDockedToolbar = {                                 ❷ 配置顶部停靠工具栏
    xtype     : 'toolbar',
    dock      : 'top',
    items     : buttons
};

var myPanel = Ext.create('Ext.panel.Panel', {
    width        : 350,
    height       : 250,
    title        : 'Ext Panels rock!',
    renderTo     : Ext.getBody(),
    html         : 'Content body',
    buttons      : {                                     ❸ 添加按钮
        items : buttons
    },
    dockedItems : [                                      ❹ 设置停靠元素
        topDockedToolbar
    ]
});
```

在代码清单4-3中，首先设置了一个可重用的按钮配置对象集合❶。这么做可以在稍后添加更多工具栏的时候轻松地渲染按钮。

然后创建一个顶部停靠的Toolbar配置对象❷。它之所以停靠在顶部，是因为设置的dock属性。后面可以看到，dock属性其他可能赋的值有bottom、left和right。

最后，渲染一个面板，给它配置一个buttons配置对象❸和一个dockedItems数组❹，该数组包含单个元素topDockedToolbar。面板在屏幕上的渲染结果应该如图4-4所示。



图4-4　带顶部停靠工具栏的面板

目前为止，除了给面板配置tbar和buttons属性以外，你还没有看到新东西。所以接下来，要添加左侧停靠、右侧停靠和底部停靠的工具栏。要把相关代码插入到面板的上方：

```
var bottomDockedToolbar = {
    xtype    : 'toolbar',
    dock     : 'bottom',
    items    : buttons
};
var leftDockedToolbar = {
    xtype    : 'toolbar',
    vertical : true,
    dock     : 'left',
    items    : buttons
};
var rightDockedToolbar = {
    xtype    : 'toolbar',
    vertical : true,
    dock     : 'right',
    items    : buttons
};
```

从这一段代码中，应该能可以看出某种模式来。dock属性决定工具栏将被停靠在哪儿。把vertical设置为true，工具栏就知道用VBox布局（垂直排列组件），而不是默认的HBox布局（水平排列组件）。

接下来要为面板的dockedItem属性添加工具栏：

```
dockedItems :[
    topDockedToolbar,
    bottomDockedToolbar,
    leftDockedToolbar,
    rightDockedToolbar
]
```

在面板渲染中用这种方法更改dockedItems的结果如图4-5所示。

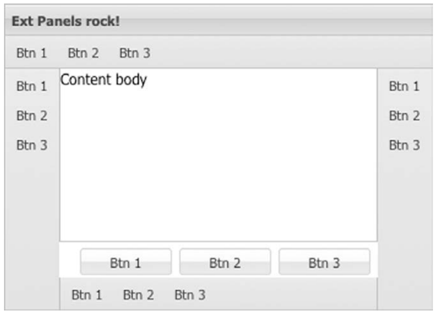


图4-5 把停靠工具栏渲染到面板的全部4个象限

在面板内容体的四面都停靠了元素，但按钮栏放错了地方。另外，我们希望底部停靠的工具栏能跟顶部停靠的工具栏宽度相同。

为了修正这些问题，必须要调整停靠元素的*权重*。

4.1.4 权重很重要

之所以在尺寸方面遇到问题，是因为Ext JS框架根据每个停靠元素的默认权重渲染了它们并设定了它们的尺寸。要了解权重的原理，可以想象重力在现实世界中是怎样的。

一个物体的重量越大，它就越接近与重力源。在Dock布局中，权重越大的组件会被渲染在越靠近内容体的地方，权重越小的停靠元素其设定尺寸的优先级越高。

最后，在下面的这个代码清单中，我们将展示完整的代码，其中包括按钮栏和4个停靠工具栏，因为要循序渐进地对代码进行改动。

代码清单4-4 带4个停靠工具栏和一个按钮栏的面板

```
var buttons = [
    { text : 'Btn 1' },
    { text : 'Btn 2' },
    { text : 'Btn 3' }
];

var topDockedToolbar = {
    xtype    : 'toolbar',
    dock     : 'top',
    items    : buttons
};

var bottomDockedToolbar = {
    xtype    :'toolbar',
    dock     :'bottom',
    items    : buttons
};

var leftDockedToolbar = {
    xtype    : 'toolbar',
    vertical : true,
```

```
        dock     : 'left',
        items    : buttons
    };

    var rightDockedToolbar = {
        xtype    : 'toolbar',
        vertical : true,
        dock     : 'right',
        items    : buttons
    };

    var myPanel = Ext.create('Ext.panel.Panel', {
        width       : 350,
        height      : 250,
        title       : 'Ext Panels rock!',
        renderTo    : Ext.getBody(),
        html        : 'Content body',
        buttons     : buttons,
        dockedItems : [
            topDockedToolbar,
            bottomDockedToolbar,
            leftDockedToolbar,
            rightDockedToolbar
        ]
    });
```

要解决的第一个问题是按钮栏的尺寸和位置设定错误。为了解决这个问题，必须改变面板的buttons属性配置方式。

```
    buttons :{
        weight : -1,
        items  : buttons
    },
```

在这里把buttons数组引用封装在一个weight属性为-1的对象里。当面板初始化时，它会识别出buttons属性是一个对象，用它创建一个工具栏的新实例，设置自定义的weight属性。因为给weight属性赋值-1，权重被视为极小，所以它将被渲染在底部停靠的工具栏下方。图4-6展现了渲染后的样子。

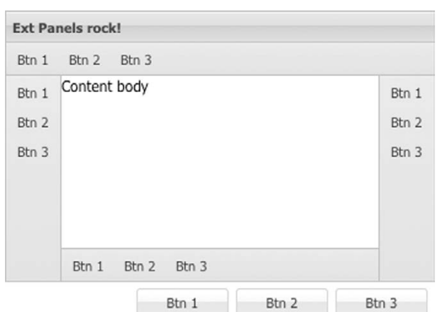


图4-6　正确渲染停靠元素

要解决的下一个问题是左侧停靠和右侧停靠的工具栏的尺寸设定问题。要加大左侧停靠和右侧停靠工具栏的权重，而不是减小底部停靠的工具栏的权重。为此要把下面的属性添加到`leftDockedToolbar`和`rightDockedToolbar`配置对象上：

```
weight  :10,
```

把新配置的`weight`属性添加到左侧停靠和右侧停靠的工具栏配置对象，可以确保底部停靠工具栏渲染时尺寸正确，如图4-7所示。可以看到，添加`weight ：10`到左侧停靠和右侧停靠的工具栏配置对象，可以让底部停靠的工具栏占据面板宽度的100%。



图4-7　修正底部停靠按钮栏

神奇的数字10？

你可能要问为什么把左侧停靠和右侧停靠工具栏的`weight`属性设为10,而不是其他数字，比如50或者100。之所以用10是因为元素停靠的每个象限都有一个默认权重。顶部是1，左侧是3，右侧是5，底部是7。之所以用10，是因为它是一个比7（底部停靠）大的约整数。

我们鼓励你多多把玩这段代码，在每个区域添加超过一个停靠元素，调整其权重。在此过程中，请记住可以往面板上停靠其他`Component`子类。

现在你有了一些关于`Panel`的经验，我们来看看它最近的子类之一：`Window`，可以用它把内容浮动在屏幕上的所有其他内容上方，替代传统的基于浏览器的弹出式窗口。

4.2　显示窗口对话框

窗口UI部件建立在面板之上，所以可以把UI组件浮动在页面上的所有其他内容上方。利用窗口可以生成一个模态对话框，屏蔽整个页面，迫使用户把焦点放在对话框上，阻止与页面上其他部分发生基于鼠标的交互。图4-8展现了怎么用这个类来获得用户的注意并请求输入。

使用`Window`类跟使用`Panel`类很像是，不同的是必须要考虑一些问题，比如是不是想关闭调整尺寸功能，或者想不想把窗口局限在浏览器视口的范围内。

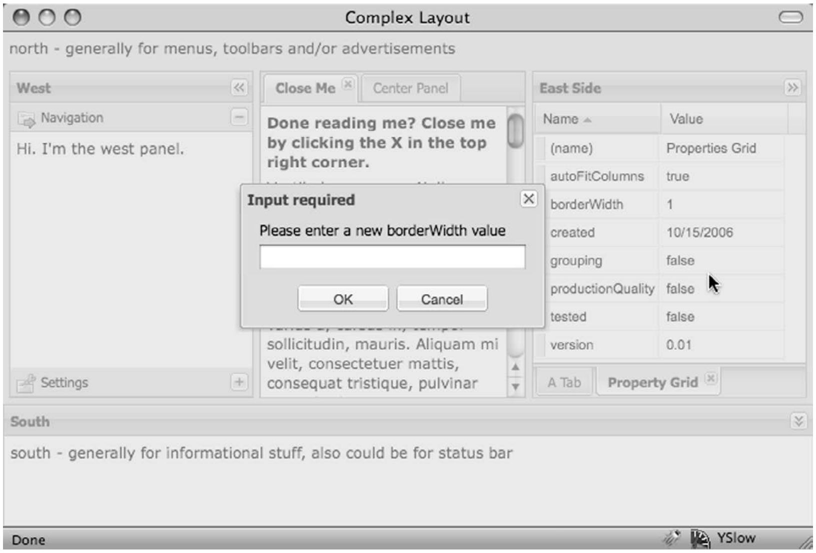


图4-8　一个Ext JS模态窗口，它屏蔽了浏览器的视口

4.2.1　构建一个窗口

我们来深究一下怎么构建一个窗口。为此需要一个没有加载部件的基础的Ext JS页面，如以下代码清单所示。

代码清单4-5　构建动画窗口

```
var win;
var newWindow = function(btn) {
    if (!win) {
        win = Ext.create('Ext.window.Window', {
            animateTarget : btn.el,
            html          : 'My first vanilla Window',
            closeAction   : 'hide',
            id            : 'myWin',
            height        : 200,
            width         : 300,
            constrain     : true
        });
    }
    win.show();
}
new Ext.Button({
    renderTo : Ext.getBody(),
    text     : 'Open my Window',
    style    : 'margin:100px',
    handler  : newWindow
});
```

❶ 创建新的窗口

❷ 约束Window实例

❸ 创建按钮

这个代码清单里的做法略有不同，为的是看到调用窗口close和hide方法的动画。要做的第一件事是创建一个全局变量win，将用它来引用即将创建的窗口。创建一个方法newWindow❶，它将成为要创建的按钮的处理程序，负责生成新的窗口。

我们先花点时间来看一下窗口的一些配置选项。通过调用窗口的show和hide方法生成动画效果的方法之一，是制定一个animateEl属性，它是对DOM的某个元素或者元素ID的引用。如果不在配置选项中指定元素，可以在调用show或者hide方法的时候指定它，它们的参数相同。在这里指定的是按钮的元素。另一个重要的配置选项是closeAction，它的默认值是close，当点击关闭工具（×）的时候销毁窗口。你不想它出现在这个实例中，所以把它设为hide，指示关闭工具调用hide方法而不是close方法。还把constrain参数❷设为了true，这指示窗口的拖放处理程序防止窗口被移动到浏览器的视口范围以外。

最后，创建一个按钮❸，点击它会调用newWindow方法，从而使窗口根据按钮的元素生成动画效果。点击关闭工具能使窗口隐藏。渲染后的结果如图4-9所示。

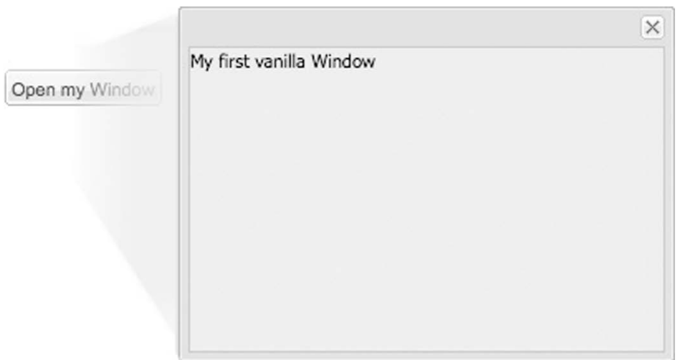


图4-9　代码清单4-3的渲染结果，你在其中创建了一个窗口，点击按钮时根据按钮的元素生成动画效果

因为在点击关闭工具的时候没有销毁窗口，所以可以重复地显示和隐藏窗口。当有必要销毁窗口的时候，可以调用它的destroy或者close方法。现在你已经有了创建可重用窗口的经验，可以开始探索其他配置选项，以进一步改变窗口的行为。

4.2.2　更多窗口配置

很多情况下需要让一个窗口执行某项操作以满足应用的要求。本节将带你学习一些常用的配置选项。

有时候要生成一个模态和刚性的窗口，为此需要设置几个配置选项，如以下代码清单所示。

代码清单4-6　创建一个刚性模态窗口

```
var win = Ext.create('Ext.window.Window', {
    height       : 75,
    width        : 200,
```

❶ 确保页面被屏蔽

```
        modal         : true,
        title         : 'This is one rigid window',
        html          : 'Try to move or resize me.I dare you.',
        plain         : true,
        border        : false,
        resizable     : false,
        draggable     : false,
        closable      : false,
        buttonAlign   : 'center',
        buttons       : [
            {
                text    : 'I give up!',
                handler : function() {
                    win.close();
                }
            }
        ]
    })
    win.show();
```

❷ 防止调整尺寸

❸ 禁用窗口移动

❹ 防止窗口关闭

在代码清单4-6中，创建了一个极其严格的模态窗口。必须设置好几个选项。第一个选项modal❶，指示窗口用一个半透明的div屏蔽页面的其他内容。然后把resizable❷设置为false，这可以防止通过鼠标操作调整窗口的尺寸。为了防止窗口被在页面上拖来拖去，把draggable❸设为false。只要一个中心按钮来关闭窗口，所以把closeable❹设为false，隐藏关闭工具。最后，设置一些修饰性参数，plain、border和buttonAlign。把plain设为true可以让内容体背景变透明。如果再把border设为false，那窗口就会呈现为一个统一的单元。因为想让那个按钮置于中央，所以把buttonAlign属性指定为'center'。渲染后的示例应该如图4-10所示。

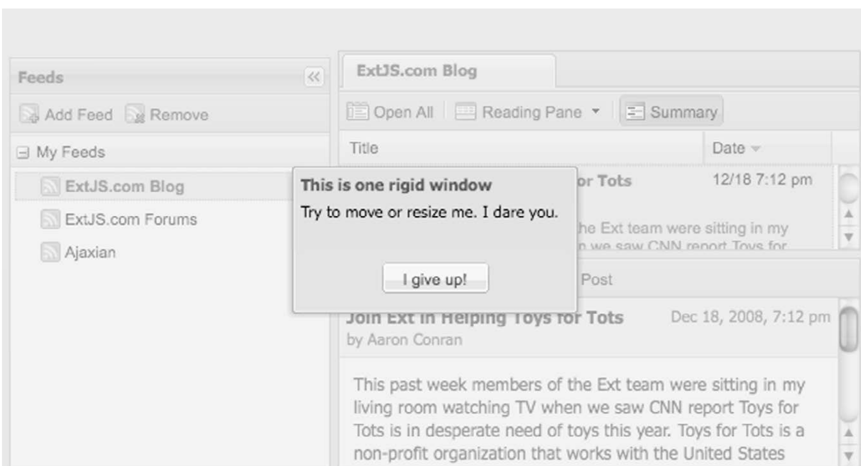


图4-10　在Ext JS SDK的feed viewer示例中渲染的第一个严格的模态窗口

还有些时候希望在窗口上放松限制。比如说，有些情况下需要一个可以调整尺寸的窗口，但不能小于特定的尺寸。为此要允许调整尺寸（resizable），并限定minWidth和minHeight参数。可惜的是，要设定窗口尺寸的上限没有捷径可走。

虽然创建你自己的窗口有很多原因，有时候需要一个快速粗暴的原因，比如说，显示一条用户数据的信息或者提示框。Window类有一个子类MessageBox可满足这一需求。

4.3 消息框

MessageBox是一个可重用而又多用途的单例类，借此可以用一个简单的方法调用来替换部分基于浏览器的常用消息框，比如alert和prompt。关于MessageBox类要了解的最重要的一点是，它并不像传统的警告框或者提示框那样阻止JavaScript执行，这我们认为是好事。在用户领会或者输入信息的同时，代码可以执行Ajax查询甚至可以操作UI。指定后的MessageBox将在窗口关闭的时候执行一个回调方法。

4.3.1 警告用户

在开始使用MessageBox类之前，让我们来创建一个回调方法。你稍后会用得着：

```
var myCallback = function(btn, text) {
    console.info('You pressed '  + btn);
    if (text) {
        console.info('You entered :' + text)
    }
}
```

myCallback方法将使用Firebug的控制台来回显任何点击的按钮和输入的文字。MessageBox只会向回调方法传入两个参数：按钮ID和输入的任意文本。既然有了回调方法，让我们来弹出一个警告消息框：

```
var msg   = 'Your document was saved successfully';
var title = 'Save status:'
Ext.MessageBox.alert(title, msg);
```

在这里调用MessageBox.alert方法，后者会弹出一个窗口（见图4-11，左图），当点击OK按钮时会关闭该窗口。如果想在关闭窗口时执行myCallback，就把它添加为第三个参数。

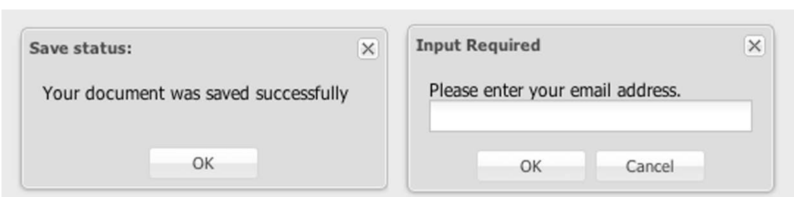


图4-11 MessageBox的警告框（左）和提示框（右）模态对话窗口

现在我们已经看过警告框，再来看看如何用MessageBox.prompt方法来请求用户输入：

```
var msg   = 'Please enter your email address.';
var title = 'Input Required'
Ext.MessageBox.prompt(title, msg, myCallback);
```

调用MessageBox.prompt方法，把回调方法的引用传进去，效果如图4-11（右）所示。输入一些文本，然后点击Cancel。在Firebug控制台中，你将看到点击的按钮ID和输入的文本。

这就是MessageBox警告框和提示框窗口。我们会发现这些窗口很方便，因为不用自行创建单例实例来生成这些UI部件。当要实施一个Window类满足需求的时候记得用它们。

我们要坦白一个小秘密。alert和prompt方法实际上都是比它们大得多而且可配置性极强的MessageBox.show方法的快捷方法。接下来这个示例将告诉你，如何使用show方法显示一个带图标的多行文本框。

4.3.2 MessageBox的高阶方法

MessageBox.show方法提供了一个接口，让你可以用24个可选项的任意组合来显示MessageBox。和之前我们学习过的快捷方法不同，show接受典型的配置对象作为其参数。让我们来展现一个带图表的多行文本框：

```
Ext.Msg.show({
    title     : 'Input required:',
    msg       : 'Please tell us a little about yourself',
    width     : 300,
    buttons   : Ext.Msg.OKCANCEL,
    multiline : true,
    fn        : myCallback,
    icon      : Ext.MessageBox.INFO
});
```

在渲染以上示例的时候，它会显示一个模态对话框，如图4-12（左）所示。接下来，我们来看看如何创建一个包含一个图表和三个按钮的警告框：

```
Ext.Msg.show({
    title     : 'Hold on there cowboy!',
    msg       : 'Are you sure you want to reboot the internet?',
    width     : 300,
    buttons   : Ext.Msg.YESNOCANCEL,
    fn        : myCallback,
    icon      : Ext.MessageBox.ERROR
});
```

以上代码示例奖显示三按钮模态警告框窗口，如图4-12（右）所示。

虽然这两个自定义MessageBox示例中的所有代码都是不言自明的，我们觉得还是很有必要着重谈一下往MessageBox公共属性中传入引用的两个配置选项。

buttons参数用于告知单例实例应该显示哪些按钮。虽然传入了一个现存属性Ext.

MesageBox.OKCANCEL的引用，也可以通过把buttons设置为一个空对象，比如{}，以不显示任何按钮。

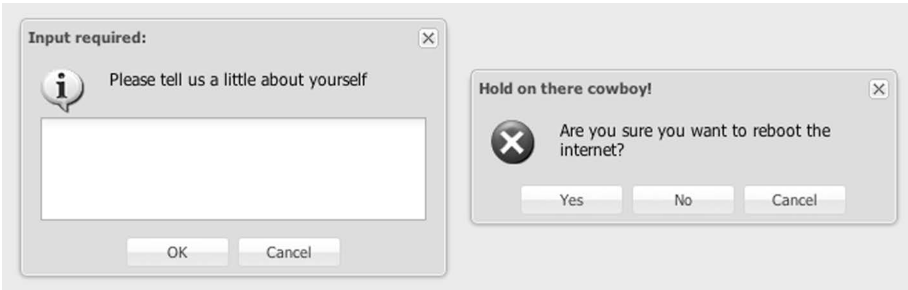


图4-12 一个带图标的多行输入框（左）和一个三按钮图标警告框（右）

如果想只显示按钮，但不自定义文本，那单例实例已经拥有一套预定义的常用组合：CANCEL、OK、OKCANCEL、YESNO和YESNOANCEL。

此外，你还可以自定义想要显示的按钮。但并不是把buttons设置为一个属性，而是设置buttonText。比如，要显示Yes和Cancel按钮，就把buttonText设置为{ yes :'Sure thing!', cancel :'No way!'}，其中对象的键是按钮的ID，而字符串是按钮的显示文本。

icon参数跟buttons参数的使用方法相同，只不过它是对一个字符串的引用。MessageBox类有三个预定义的值：INFO、QUESTION和WARNING。它们是对身为CSS类的字符串的引用。如果希望显示你自己的图标，就创建自己的CSS类，并把自定义CSS类的名称作为icon参数传入。以下是一个自定义CSS类的示例：

```
.icon-add {
    background-image: url(/path/to/add.png) !important;
}
```

现在你已经亲身尝试了一些MessageBox的高阶方法，我们可以探索如何用MessageBox来显示一个动画对话框了，后者可以给用户提供关于某一特定过程的实时最新信息。

4.3.3 显示一个动画式等待对话框

当需要停止一个特定的工作流时，必须显示某种模态消息框，它可以非常简单枯燥，就像一个带有“Please wait”消息的模态对话框。我们希望能给这个应用增加一点趣味，设计一个动画式的“等待”对话框。可以利用MessageBox类创建一个看似毫不费力的无限循环进度条：

```
Ext.MessageBox.wait("We're doing something...", 'Hold on...');
```

这条代码会生成一个等待对话框，如图4-13所示（左）。如果这里的语法看似有点奇怪，那是因为第一个参数是消息主体文本，而第二个参数是标题。这跟警告框和提示框的调用正好相反。如果想在动画进度条本身的主体内显示文本，可以把单个文本属性作为第三个参数传入，比如

{text:'loading your items'}。图4-13（右）展现了把进度条文本添加到虚拟等待对话框里的样子。



图4-13　一个简单的动画MessageBox等待对话框，进度条以一个预设的固定时间间隔进行无限循环（左），以及一个进度条中有文本的类似的消息框（右）

虽然这乍一看似乎很酷，但其实没有交互，因为文本是静态的，并没有控制进度条的状态。可以使用方便的show方法并传入一些参数，来自定义等待对话框。使用这个方法就可以有足够的余地来随时随地更新进度条的进度。要创建一个自动更新的等待对话框，必须创建一个相当复杂的循环（如以下代码清单所示），所以请继续看下去。

代码清单4-7　创建一个动态更新的进度条

```
Ext.MessageBox.show({
    title        : 'Hold on there cowboy!',
    msg          : "We're doing something...",
    progressText : 'Initializing...',
    width        : 300,
    progress     : true,
    closable     : false
});
var updateFn = function(num){
    return function(){
        if(num == 6){
            Ext.MessageBox.updateProgress(100,
                 'All Items saved!');
            Ext.Function.defer(Ext.MessageBox.hide,
                 1500, Ext.MessageBox);
        }
        else{
            var i = num/6;
            var pct = Math.round(100 * i);
            Ext.MessageBox.updateProgress(i,
                  pct + '% completed');
        }
    };
};
for (var i = 1; i < 7; i++){
    setTimeout(updateFn(i), i * 500);
}
```

❶ 显示进度条

❷ 更新百分比、文本

❸ 添加循环数0.5秒时间间隔

在代码清单4-7中创建一个消息框，progress选项❶设为true，会显示进度条。然后定义一个非常复杂的更新器函数，名为updateFn，并以一个预定义的时间间隔进行调用。在这个函数中，如果传入的数字等于上限6，就把进度条更新至100%宽，并显示完成文本。还把消息框的

关闭推迟了1.5秒。否则要计算一个完成百分比，并相应地更新进度条的宽度和文本❷。最后创建一个循环，连续6次调用`setTimeout`❸，而这将推迟循环次数乘以0.5秒的时间调用`updateFn`。这个很长的示例运行结果如图4-14所示。稍加努力，就可以让用户在继续下一步操作前动态获取当前的运行状态。



图4-14　自动更新等待消息框（左），和在自动关闭前最后更新的同一等待消息框（右）

4

本节，你学习了如何创建弹性而又极其刚性的窗口来获得用户的关注，还探索了使用Ext JS的超单例类Ext JS `MessageBox`的几种方法。先让我们把注意力放到`TabPanel`类上，它会为我们提供一个办法让UI可以包含许多屏幕的内容，但每次只显示其中之一。

4.4　组件也可以存活在标签面板中

`TabPanel`类建立在`Panel`类上，以创建一个健壮的标签界面，它让用户可以选择与某一标签相关联的任何屏幕或者UI控件。

标签面板内的标签可以是不可关闭的、可关闭的、禁用的，甚至是隐藏的，如图4-15所示。



图4-15　探索顶置和底置标签

和其他标签界面不同，Ext JS标签面板只支持顶置和底置标签条配置。这主要是因为很多浏览器仍然不支持CSS，所以垂直文本无法实现。标签面板使用Card布局，它可以快速渲染复杂的UI，因为它使用了一种称为懒惰或延时渲染的常用方法来渲染其子组件。标签面板的延时渲染功能由`deferredRender`参数控制，其默认值是`true`。

延时渲染意味着只有只有渲染过的标签才会被激活。对标签面板来说，拥有多个有复杂UI控件的子元素是很平常的，比如图4-16里的标签面板，渲染它需要大量的CPU时间。延迟对于每个子元素的渲染直到其被激活，可以加速标签面板的初步渲染，使用户拥有一个具有更高响应性的部件。

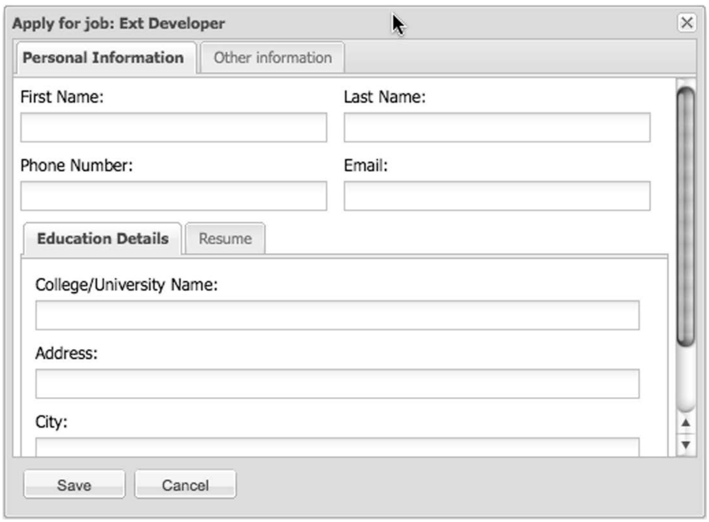


图4-16　一个带有多个布局复杂的子元素的标签面板

现在你要构建第一个标签面板了。

4.4.1　构建第一个标签面板

`TabPanel`是`Panel`的一个直接子类，灵活运用了`CardLayout`。标签面板的主要功能是管理标签条中的标签。标签面板子组件的管理是由`Container`类承担的，而布局管理是由`CardLayout`承担的。下面的代码清单展现了如何构建你的第一个标签面板。

代码清单4-8　探索一个标签面板

```
var simpleTab = {                                    ❶ 引入静态标签
    title  : 'Simple tab',
    html   : 'This is a simple tab.'
};

var closableTab = {                                  ❷ 创建可关闭标签
    title    : 'I am closable',
```

```
        html     : 'Please close when done reading.',
        closable : true
    };
    var disabledTab = {                                    ❸ 添加禁用的
        title    : 'Disabled tab',                            标签
        itemId   : 'disabledTab',
        html     : 'Peekaboo!',
        disabled : true,
        closable : true
    };

    var tabPanel = Ext.create('Ext.tab.Panel', {          ❹ 实例化标签
        activeTab : 0,                                        面板
        itemId    : 'myTPanel',
        items     : [
            simpleTab,
            closableTab,
            disabledTab
        ]
    });

    Ext.create('Ext.window.Window', {                      ❺ 渲染标签面板
        height :300,
        width  :400,
        layout :'fit',
        items  : tabPanel
    }).show();
```


虽然在这段代码中本可以用单个大型对象来定义所有元素，但我们觉得最好还是把它拆分开来，以求清晰。前三个变量用通用对象的形式定义了标签面板的子元素，前提是假定`TabPanel`类的`defaultType`（XType）是`Panel`。第一个子元素是一个简单的不可关闭标签❶。这里值得注意的是所有标签默认都是不可关闭的。这就是为什么第二个标签❷把`closable`设为了`true`。接下来，有一个可关闭和禁用的标签。每个这些子面板配置对象都有自己的`itemId`，这样就可以定位它们，并执行某些操作，比如启用、隐藏和禁用。

然后继续实例化标签面板❸。把`activeTab`参数设置为`0`。这么做是因为希望第一个标签在标签面板❹渲染后就激活。可以在标签面板的`items`混合集合中指定任何索引号。因为该混合集合是一个数组，所以第一个元素的索引总是`0`。最后，标签面板的`items`数组制定了三个标签。

接下来，为标签面板创建一个容器，一个`Ext.Window`的实例❺。为窗口指定一个Fit布局，并把标签面板的引用设置为它的唯一元素。代码渲染后生成的标签面板如图4-17所示。

现在你已经渲染了第一个标签面板，接下来可以开始体验它的乐趣了。你可能已经关闭了“I am closable”标签，这没问题。如果还没有关闭，那大可以尽情地探索渲染后的UI控件，并在你觉得可以的时候关闭这个唯一的可关闭标签，这样就剩下了两个标签：“My first tab”和“Disabled tab”。



图4-17　在窗口内渲染的第一个标签面板

4.4.2　你应该知道的标签管理方法

因为TabPanel类是Container的一个子类，所以所有通用的子元素管理方法都可以使用。其中包括add、remove和insert。不过你还需要了解另外的几个方法，以充分利用TabPanel类。

首先是setActiveTab，它可以激活一个标签，就好像用户已经在标签条上选择了该元素，并接受该标签的索引或者组件ID：

```
var tPanel = Ext.ComponentQuery.query('#myTPanel')[0];
tPanel.add({
    title  : 'New tab',
    itemId : 'myNewTab',
    html   : 'I am a new Tab'
});
tPanel.setActiveTab('myNewTab');
```

执行这段代码将会生成一个标题为“New tab”的新标签，它会自动激活。在add操作后调用setActiveTab类似于在一个通用容器上调用doLayout。还可以在代码运行的同时启用和禁用标签，但这需要另辟蹊径，而不只是简单调用标签的方法。

标签面板没有enable和disable方法，所以要启用或者禁用一个子元素，必须调用子元素本身的相应方法。可以使用代码清单4-8来启用禁用了的标签。可以使用不久前创建的tPanel引用，搜索禁用了的子元素并启用，代码如下：

```
tPanel.down('#disabledTab').enable();
```

是的，仅此而已。相应的标签条元素（标签UI控件）反映该元素不再被禁用。这是因为标签面板授权其子元素的（你猜得没错）enable和disable事件管理其相关的标签条元素。

除了启用和禁用标签以外，你还可以隐藏它们。要隐藏标签，就必须访问标签面板子元素的

tab属性。为了加以演示，要隐藏禁用了的标签，然后再显示：

```
tPanel.down('#disabledTab').tab.hide();
```

要使之重新显示，执行以下代码：

```
tPanel.down('#disabledTab').tab.show();
```

你现在就知道创建和管理一个标签面板有多容易了。

4.5 小结

我们讨论了瑞士军刀式的UI显示部件面板的很多内容，这已经足以让任何一名开发人员晕头转向了。在探索Panel类的时候，我们看到它提供了众多选项来显示用户交互内容，其中包括工具栏、按钮、标题栏图标以及微型工具。

你把Window类用作一个通用容器，并掌握了动态添加和删除子元素的技巧，从而有能力动态地大幅度改变一个完整的UI，或者单个部件或控件。

在探索Window类和它的近亲MessageBox的时候，你了解了如何替换掉通用的警告和提示对话框，以获取用户的注意、显示信息或者请求用户输入。你还把玩了一下动画式的等待消息框MessageBox。

最后，你探究了标签面板，学习了如何动态管理标签元素，了解了该UI控件带来的几个可用性缺陷。

在接下来的一章中，你将探究很多Ext JS布局方案，并将了解这些控件的通常用法和缺陷。

第 5 章 探究布局

5

本章内容

- ❑ 使用布局系统
- ❑ 探究`Layout`类继承模型
- ❑ 了解Card布局

在构建应用的时候，很多开发人员都纠结于如何组织UI，以及该使用哪些工具来组织。本章，你将获得必需的经验，可以用一种训练有素的方式做这些决定。我们将从介绍那些Ext JS 4的新组件布局开始，然后再去探究为数众多的容器布局模型，并确定那些最好的方法以及常见问题。

容器布局管理方案负责组织排列屏幕上的部件。其中包括简单的布局方案，比如Fit布局，调整容器单个子项的尺寸以适合该容器的内容体；还有复杂的布局，比如Border布局，它把容器的内容体分割成5个可管理的分片或者区域。

我们要对容器布局进行不吝篇幅的探讨，并给出一些较长的示例，以此为跳板来设计使用自己的布局。但在我们继续深入描述每一个容器布局管理方案以前，先来聊聊布局管理器是如何工作的，并介绍新的组件布局。

5.1 布局管理器如何工作

之前提到过，布局管理方法负责组织排列屏幕上的部件。为此，它们要跟踪每一个子项与其他子项的相对位置。排列部件采用的策略取决于你使用的那个布局管理器。布局管理器分为两组：组件布局和容器布局。

5.1.1 组件布局

组件布局是Ext JS 4的新功能，被用于布置组件的内部元素。在日常的使用中，你应该很熟悉Dock组件布局。Dock布局负责管理诸如工具栏的可停靠元素，让你可以有选择性地添加多个顶部和底部工具栏，以及左侧和右侧工具栏（如果想复习一下有关停靠的细节，请回顾4.2节）。

如果要实现自己的组件，以及组件布局管理方案，那我们建议熟悉一下组件布局现有的层级

关系，并选择一个相关的类进行扩展。

如果你一直在使用Ext JS 3和更早的版本，那或许还记得Form布局用起来有多麻烦多复杂。在Ext JS 4中再也不需要Form布局，因为引入了Field组件布局及其子类，以及代码库中的相关功能。

记住所有标准组件已有各自对应的组件布局，所以要完成日常编程，并不需要详细了解每一个组件布局。本章中，我们把注意力放在容器布局上。

5.1.2 容器布局

借助容器布局可以管理一个容器内的子组件的位置和尺寸。当在一个容器上添加或者删除一个组件时，容器会与其父容器沟通，根据布局管理器方案调整其姊妹容器或组件的尺寸。

所有容器布局管理器都共享`Ext.layout.container.Container`现有的通用功能。我们将详细探究每个布局管理器。

在容器布局中我们要首先来看一看Auto布局，它是容器的默认布局。Auto布局是最基本的布局管理器，它与容器布局共享通用功能。

5.2 Auto 布局

你可能还记得Auto布局是任何容器示例的默认布局。它把元素上下叠加地放置在屏幕上。虽然Auto布局不显式调整子项尺寸，但如果内容体没有被限定的话，一个子项的宽度可能与该容器的内容体宽度一致。

实现Auto布局是最容易的，只需要添加和删除子项。为此需要用好几个组件来设置一个动态示例，如以下代码清单所示。完成后，布局将如图5-1所示。

代码清单5-1　实现Auto布局

```
var childPnl1 = {                                        ❶ 配置第一个子面板
    frame  : true,
    height : 50,
    html   : 'My First Child Panel',
    title  : 'First children are fun'
};
var childPnl2 = {                                        ❷ 配置第二个面板
    width  : 150,
    html   : 'Second child',
    title  : 'Second children have all the fun!'
};
var myWin = Ext.create("Ext.Window", {                   ❸ 创建窗口
    height     : 300,
    width      : 300,
    title      : 'A window with a container layout',
    autoScroll : true,                                   ❹ 设置可滚屏
    items      : [                                       ❺ 添加子面板
        childPnl1,
        childPnl2
    ],
```

```
        tbar :[
            {
                Text    : 'Add child',
                handler : function() {
                    var numItems = myWin.items.getCount() + 1;
                    myWin.add({
                        title       : 'Child number ' + numItems,
                        height      : 60,
                        frame       : true,
                        collapsible : true,
                        collapsed   : true,
                        html        : 'Yay, another child!'
                    });
                }
            }
        ]
    });
    myWin.show();
```

❻ 配置工具栏

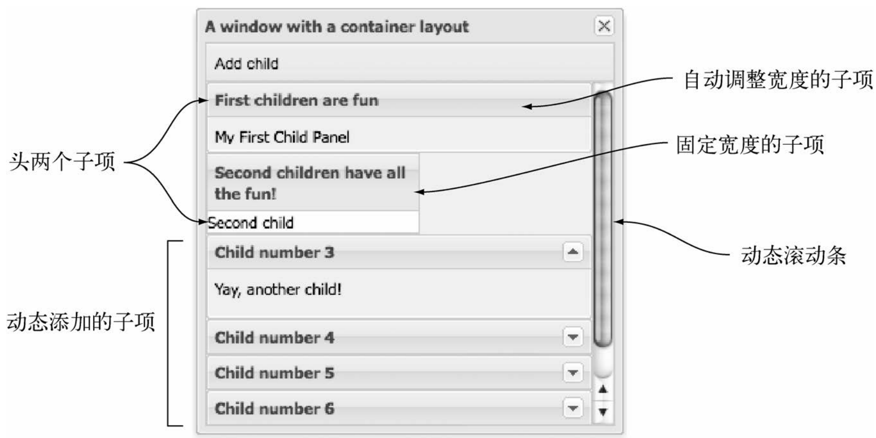


图5-1　首次实现Auto布局的结果

在代码清单5-1中，首先要做的是用Xtype给两个子项实例化对象引用，并由一个窗口来加以管理。childPnl1❶和childPnl2❷。这两个子项是静态的。接下来，初始化myWin❸引用，它是一个Ext.Window示例。还要把autoScroll属性❹设为true。这指示容器把CSS属性overflow-x和overflow-y设为auto，让浏览器只有在必要的时候才显示滚动条。

请注意把子项items属性❺设为了一个数组。任何容器的items属性都可以是一个用来列举多个子项的数组示例，或者是一个单独子项的对象引用。窗口包含一个工具栏❻，上面有单个按钮，点击时会往窗口添加一个动态元素。请注意在Ext JS 4之前，可以在删除或者添加一个元素之后调用父容器的doLayout，现在不需要这样做了，因为在组件/容器的层级关系中是双向沟通的。在早期版本中，添加一个或者多个子项后你会调用myWin.doLayout。如果要对组件进行批量更新，那可以在容器上把suspendLayout设为true以避免调用doLayout。渲染后的窗口应

该如图5-1所示。

虽然Auto布局没有提供多少管理子项尺寸的方法，但它并非完全无用。相比于它的子类它很简洁，所以如果想展现有固定尺寸的子项，用它最合适。不过有些时候，你希望动态调整子项的尺寸来适应容器的内容主体。这时候Anchor布局就有用了。

5.3　Anchor 布局

Anchor布局跟其他容器布局很相似，因为它把子项上下叠在一起，但它会通过在每个子项上指定的anchor参数来进行动态尺寸调整。这个anchor参数用于计算子项相对于父容器内容体的尺寸，要么指定为一对百分比，要么被指定为一对整数偏移值。anchor参数是一个使用以下格式的字符串：

```
anchor :"width, height" // 或者"width height"
```

图5-2展现了构建结果。

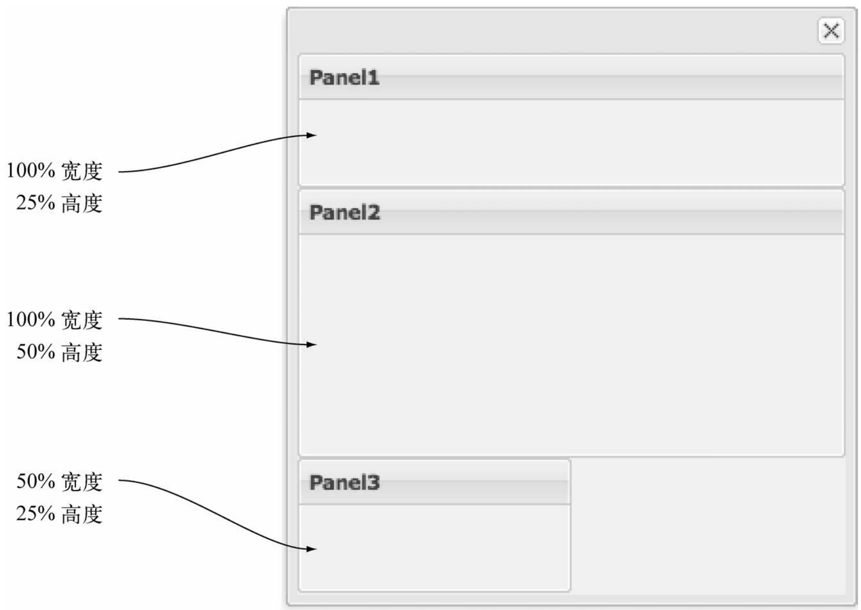


图5-2　在代码清单5-2中首次实现Anchor布局的渲染结果

在下面的代码清单中，将首次尝试用百分比来实现一个Anchor布局。

代码清单5-2　用百分比设置的Anchor布局

```
var myWin = Ext.create("Ext.Window", ({          ◁ ❶ 创建窗口
    height      : 300,
    width       : 300,
    layout      : 'anchor',          ◁ ❷ 设置布局
    border      : false,
```

```
    anchorSize : '400',
    items      : [
        {
            title  :'Panel1',
            anchor :'100%, 25%',          ❸ 设置尺寸
            frame  : true
        },
        {
            title  :'Panel2',
            anchor :'0, 50%',             ❹ 配置尺寸
            frame  : true
        },
        {
            title  :'Panel3',
            anchor :'50%, 25%',           ❺ 设置大小
            frame  : true
        }
    ]
}));
myWin.show();
```

在代码清单5-2中，实例化一个`Ext.WIndow`的示例`myWin`❶，并将布局指定为`'anchor'`❷。第一个子项`Panel1`，其`anchor`参数❸被指定为父容器宽度的100%，和父容器高度的25%。`Panel2`其`anchor`参数❹略有不同，`width`参数是`0`，也就是100%简单表达方式。`Panel2`的`height`设为父容器高度的50%。`Panel3`的`anchor`参数❺被设为50%的相对`width`和25%的相对`height`。渲染后的效果应该如图5-2所示。

用百分比设定相对尺寸是不错，但也可以指定偏移值，这使得Anchor布局有更大的灵活性。偏移值是根据内容体尺寸加上偏移值来计算的。通常来说，这里指定的偏移量是负数，以确保子项在视野范围内。让我们来回顾一下代数知识，加一个负整数就相当于减去一个绝对整数。如果指定一个正偏移值会导致子项的尺寸比内容体更大，那就需要一个滚动条。

我们将使用之前的示例来探究偏移量，只对代码清单5-2中的子项XType进行改动：

```
items :[
    {
        title  : 'Panel1',
        anchor : '-50, -150',
        frame  : true
    },
    {
        title  : 'Panel2',
        anchor : '-10, -150',
        frame  : true
    }
]
```

经历上述布局改动后渲染好的面板应该如图5-3所示。我们把子项的数量减少到了两个，以便更容易展现偏移值如何使用，以及它们为何可能会给你惹不少麻烦。

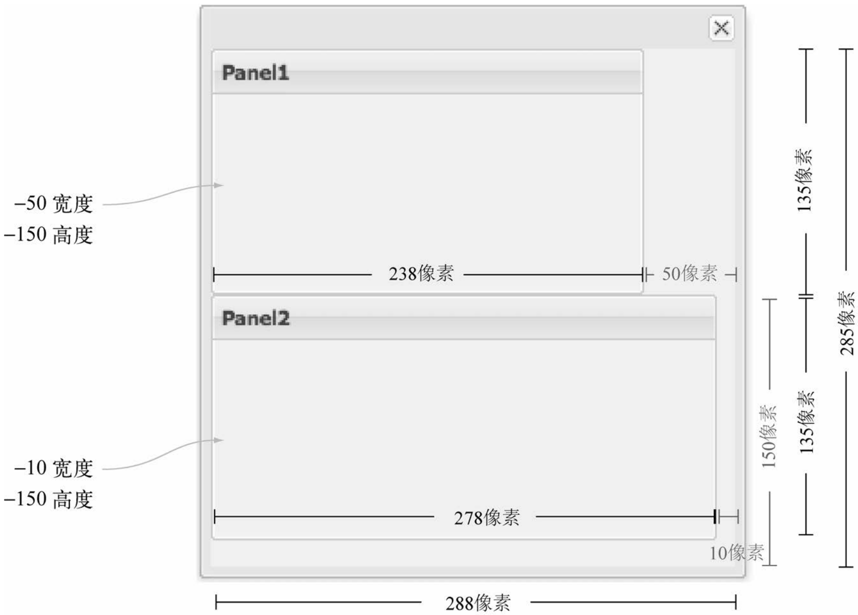


图5-3 用偏移值通过尺寸计算来设置一个Anchor布局

解析这里的情况很重要，而这需要做一点数学计算。用Firebug来检视DOM，你会发现该窗口的内容体高285像素、宽288像素。通过简单的计算，可以判断出`Panel1`和`Panel2`的尺寸应该是多少：

```
Panel1 Width  = 288px - 50px  = 238px
Panel1 Height = 285px - 150px = 135px
Panel2 Width  = 288px - 10px  = 278px
Panel2 Height = 285px - 150px = 135px
```

可以看出显然两个子面板在窗口内完美契合。如果加上两个面板的高度，也可以看出它们契合，高度相加只有270像素。但是如果在垂直方向上调整尺寸呢？注意到什么怪事吗？把窗口的高度增加超过15像素，就会导致`Panel2`被推到屏幕以外，同时在`windowBody`中显示滚动条。

请记住使用这种布局，子项的尺寸是父容器内容体尺寸的相对值加上一个常数，也就是偏移量。为了解决这个问题，可以混合使用固定尺寸和偏移值。为了探究这个概念，更改`Panel2`的`anchor`参数，并加上一个固定高度：

```
{
    title  : 'Panel2',
    height : 150,
    anchor : '-10',
    frame  : true
}
```

这个改动使得Panel2的高度固定在150像素。新渲染的窗口现在可以调整到几乎任何尺寸，Panel1会增长到窗口内容体高度减去150像素，这样留出的垂直空间，刚好可以让Panel2停留在屏幕内。这样做的一个好处是Panel2的宽度还是相对的。

Anchor布局被用于多种不同的布局任务。Anchor布局默认是由Ext.form.Panel类使用，但它也可以用于任何容器或者任何可以容纳其他子项的子类，比如Panel或者Window。

有些时候需要完全掌控部件布局的定位。Absolute布局可完美满足这种需求。

5.4　Absolute 布局

Absoute布局仅次于Auto布局，是目前为止最简单易用的布局之一。它固定一个子项的位置，具体是通过把子项的CSS'position'属性设置为'absolute',并把top和left属性设置为你在子项上设置的x和y参数。很多设计者用CSS把HTML元素置为一个position:absolute，但Ext JS利用JavaScript的DOM操作机制把属性设置为元素自身，而不用非得去动CSS。图5-4展现要构建的布局。下面的代码清单展现了如何用Absolute布局来创建一个窗口。

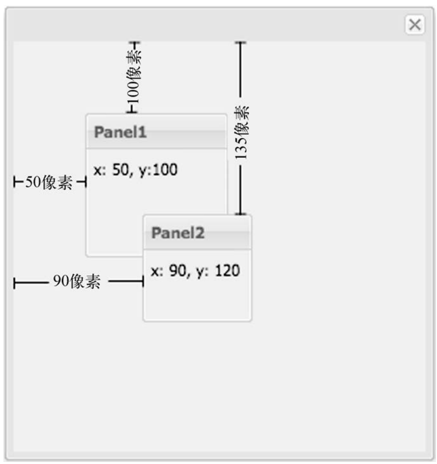


图5-4　代码清单5-3中实现的Absolute布局

代码清单5-3　Absolute布局实用示例

```
var myWin = Ext.create("Ext.Window", {
    height     : 300,
    width      : 300,
    layout     : 'absolute',          ❶ 设置布局
    autoScroll : true,
    border     : false,
    items      : [
        {
          title : 'Panel1',
          x     : 50,                 ❷ 设置子项坐标
```

```
            y      : 50,
            height : 100,
            width  : 100,
            html   : 'x: 50, y:50',
            frame  : true
          },
          {
            title  : 'Panel2',           ❸ 设置子项坐标
            x      : 90,
            y      : 120,
            height : 75,
            width  : 100,
            html   : 'x: 90, y: 120',
            frame  : true
          }
      ]
    });
    myWin.show();
```


现在，这段代码中的大部分你应该都觉得熟悉了，但也有一些新的参数。第一个值得注意的改变是窗口的`layout`❶参数被设置为`'absolute'`。往窗口添加两个子项。因为用的是Absolute布局，所以需要指定*X*坐标和*Y*坐标。

第一个子项`Panel1`的*X*❷（CSS的`left`属性）坐标设置为50像素，而*Y*（CSS的`top`属性）坐标设置为50像素。第二个子项`Panel2`的*X*❸和*Y*坐标分别设置为90像素和120像素。渲染后的代码应该如图5-4所示。

这个示例里一个很明显的细节是`Panel2`交叠在`Panel1`上。`Panel2`之所以在上面，是因为它在DOM树里的位置。`Panel2`的元素在`Panel1`的元素上方，而且因为`Panel2`的CSS位置属性也被设为`'absolute'`，它将会显示在`Panel1`上方。当你在实现这个布局的时候一定要时刻记住发生交叠的风险。另外，因为子项的位置是固定的，Absolute布局对能调整尺寸的父容器而言并非理想的解决方案。

如果有一个子项，想让它随着父容器一起调整尺寸，那Fit布局就是最佳选择。

5.5 Fit 布局

Fit布局迫使一个容器的单个子项填充它的内容体元素，也是一种特别简单的布局。图5-5展现了使用这种布局的最终结果，具体使用方法见以下代码清单。

代码清单5-4 Fit布局

```
var myWin = Ext.create("Ext.Window", {
    height     : 200,
    width      : 200,                 ❶ 配置布局
    layout     : 'fit',
    border     : false,
    items      : [
      {
        title : 'Panel1',              ❷ 添加子项
```

```
            html   : 'I fit in my parent!',
            frame  : true
        }
    ]
});
myWin.show();
```



图5-5　使用Fit布局（代码清单5-4）

在代码清单5-4中把窗口的`layout`属性设置为`'fit'`❶，并实例化了单个子项，一个`Ext.Panel`的示例❷。该子项的XType根据窗口的`defaultType`属性预设，后者由窗口的原型自动设置为`'panel'`。渲染后的面板应该如图5-5所示。

Fit布局让有一个子项的容器获得无缝外观的好方法。不过一个容器里经常会有多个部件。所有其他布局管理方案通常都是用来管理多个子项的。Accordion布局是其中最美观的布局之一，用它可以垂直地堆叠可折叠的元素，每次只给用户显示一个元素。

5.6　Accordion 布局

在以下代码清单中使用的Accordion布局，是VBox布局的一个直属子类。当想显示垂直堆叠的多个面板，但只有一个面板可以被展开或收缩时，这种布局很有用。图5-6展现了最后的结果。

代码清单5-5　Accordion布局

```
var myWin = Ext.create("Ext.Window", {
    height        : 200,
    width         : 300,
    border        : false,
    title         : 'A Window with an Accordion layout',
    layout        : 'accordion',
    layoutConfig : {
        animate  : true
    },
    items         : [
        {
```

❶ 创建委托实例

❷ 配置布局

❸ 添加首个子项

```
                xtype          : 'form',
                title          : 'General info',
                bodyStyle      : 'padding: 5px',
                defaultType    : 'field',
                fieldDefaults  : {
                    labelWidth : 50
                },
                labelWidth  : 50,
                items       : [
                    {
                      fieldLabel : 'Name',
                      anchor     : '-10'
                    },
                    {
                      xtype      : 'field',
                      fieldLabel : 'Age',
                      size       : 3
                    },
                    {
                      xtype      : 'combo',
                      fieldLabel : 'Location',
                      anchor     : '-10',
                      store      : [ 'Here', 'There', 'Anywhere' ]
                    }
                ]
              },
              {                                              ❹ 创建多行
                xtype  : 'panel',                              文本框
                title  : 'Bio',
                layout : 'fit',
                items  : {
                    xtype : 'textarea',
                    value : 'Tell us about yourself'
                }
            },
            {                                                    ❺ 添加带工具
                title : 'Instructions',                            的面板
                html  : 'Please enter information.',
                tools : [
                    {id : 'gear'}, {id:'help'}
                ]
            }
        ]
    });
    myWin.show();
```

代码清单5-5展现了Accordion布局有多好用。要做的第一件事是实例化一个窗口myWin，后者的layout属性被设为'accordion'❶。有一个配置选项你至今还没有见到过，那就是layoutConfig❷。有些布局方案有特定的配置选项，可以把它们定义为组件构造函数的配置选项。

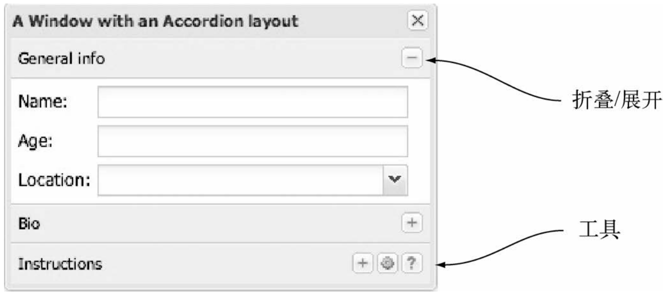


图5-6　Accordion布局是把多个元素作为单个可视化组件展现给用户的绝佳方法

这些`layoutConfig`参数可以改变一个布局的行为或者功能。在这里，为Accordion布局设置好`layoutConfig`，指定`animate:true`，也就是指示Accordion布局以动画方式呈现子项的折叠和展开。另一个改变布局行为的配置选项是`activeOnTop`，如果它被设为`true`，就会把当前激活的空间移动到栈的顶部。当第一次运用一个布局的时候，我们建议你查询它API中所有可用的选项。

接下来开始定义子项，这一步建立在你之前学到的一些知识之上。第一个子项是`FormPanel`❸，它用到本章前面的`anchor`参数。接下来指定一个面板❹，该面板的`layout`属性设为`'fit'`并且包含一个子`TextArea`。然后把最后一个子项❺定义为一个带部分工具的普通面板。渲染后的代码应该如图5-6所示。

配置布局的另一种方法

我们可以不同时使用`layout` (`String`)和`layoutConfig` (`Object`)配置，而是把`layout`配置设置为一个既包含布局类型又包含该布局任何选项的`Object`。举例来说：

```
layout :{
    type    : 'accordion',
    animate : true
}
```

很重要的一点是要注意Accordion布局只能配合`Ext.panle.Panel`和它的两个子类，`Ext.grid.Panel`和`Ext.tree.Panel`才能正常发挥作用。这是因为`Panel`（和两个特定的子类）有Accordion布局正常工作所需的东西。如果在Accordion布局内还需要什么别的元素，比如标签面板，那就用一个面板封装该元素，并把那个面板作为Accordion布局容器的一个子项添加。

虽然Accordion布局是在屏幕上同时显示多个面板的好办法，但它有局限性。比如说，如果你需要在一个容器里放置10个组件怎么办？每个元素标题栏高度的总和会占据很多宝贵的屏幕空间。Card布局可以完美满足此类需求，因为它让你可以显示或隐藏多个子组件，或者在它们之间切换。

5.7　Card 布局

Card布局确保容器的子项与容器的尺寸相一致。和Fit布局不同，Card布局可以控制多个子项。这个工具让你有充分的灵活性来创建类似向导界面的组件。

除了一开始激活的项以外，Card布局通过其公开的`setActiveItem`方法让最终开发人员实现翻页功能。为了创建类似向导界面的界面，需要创建一个方法来控制翻页：

```
var handleNav = function(btn) {
    var activeItem  = myWin.layout.activeItem,
        index       = myWin.items.indexOf(activeItem),
        numItems    = myWin.items.getCount(),
        indicatorEl = Ext.getCmp('indicator').el;

    if (btn.text == 'Forward' && index < numItems - 1) {
        index++;
        myWin.layout.setActiveItem(index);
        index++;
        indicatorEl.update(index + ' of ' + numItems);
    }
    else if (btn.text == 'Back' && index > 0) {
        myWin.layout.setActiveItem(index - 1);
        indicatorEl.update(index + ' of ' + numItems);
    }
}
```


在这里通过确定激活该项的索引号，并根据是点击"前进"还是"后退"按钮设置激活元素，来控制卡片快翻。然后更新底部工具栏的提示文本。接下来，我们来实现Card布局。以下代码清单很长也很复杂，所以请耐心看完。

代码清单5-6　Card布局实用示例

```
var myWin = Ext.create("Ext.Window", {
    height       : 200,
    width        : 300,
    border       : false,
    title        : 'A Window with a Card layout',
    layout       : 'card',
    activeItem   : 0,
    defaults     : { border : false },
    items        : [
       {
          xtype       : 'form',
          title       : 'General info',
          bodyStyle   : 'padding: 5px',
          defaultType : 'field',
          labelWidth  : 50,
          items       : [
             {
                fieldLabel : 'Name',
                anchor     : '-10',
             },
```

❶ 设置Card布局（指向 layout : 'card'）

❷ 配置激活元素（指向 activeItem : 0）

```
            {
               xtype      : 'numberfield',
               fieldLabel : 'Age',
               size       : 3
            },
            {
               xtype      : 'combo',
               fieldLabel : 'Location',
               anchor     : '-10',
               store      : [ 'Here', 'There', 'Anywhere' ]
            }
         ]
      },
      {
         xtype  : 'panel',
         title  : 'Bio',
         layout : 'fit',
         items  : {
             xtype : 'textarea',
             value : 'Tell us about yourself'
         }
      },
      {
         title : 'Congratulations',
         html  : 'Thank you for filling out our form!'
         }
      ],
      dockedItems : [
          {
              xtype : 'toolbar',
              dock  : 'bottom',
              items : [
                  {
                      text    : 'Back',            ❸ 添加导航按钮
                      handler : handleNav
                  },
                  '-',
                  {
                      text    : 'Forward',
                      handler : handleNav
                  },
                  '->',
                  {
                      type  : 'component',
                      id    : 'indicator',          ❹ 添加指示器组件
                      style : 'margin-right: 5px',
                      html  : '1 of 3'
                  }
              ]
          }
      ]
});
myWin.show();
```

代码清单5-6详细记录了如何用Card布局创建一个窗口。虽然大多数代码你应该都熟悉了，但我们还是要指出其中的几点。第一显然要注意的地方是`layout`属性❶，它被设为`'card'`。接下来是`activeItem`属性❷，容器在渲染时间把它传给布局。你把它设为`0`，指示布局在容器渲染的时候调用子组件的`render`方法。

接下来定义底部工具栏，其中包含了Forward（前进）和Back（后退）❸按钮，它们调用之前定义的`handleNav`方法和一个用来显示当前激活元素索引号的通用组件❹。渲染后的容器应该如图5-7中的容器所示。

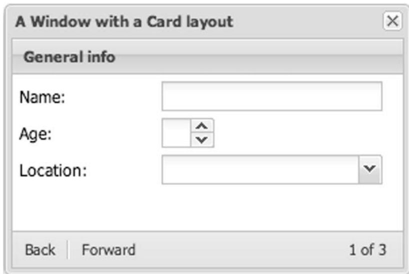


图5-7　用一个全交互式导航工具栏实现的首个Card布局（代码清单5-6）

点击Forward或Back按钮会调用`handleNav`方法，该方法会处理翻页并更新指示器组件。请记住使用Card布局时，激活元素切换的逻辑是完全由终端开发人员来创建和管理的。

除了之前讨论的这些布局，Ext JS还提供了另外几种布局方案。Column布局是UI开发人员最喜欢的布局方案之一，用于组织UI列，列宽可以达到其父容器的全宽。

5.8　Column 布局

把组件组织为列，可以在一个容器上并排显示多个组件。和Anchor布局一样， Column布局可以设置子组件的绝对宽度或者相对宽度。在使用这种布局时有一些事要注意。我们稍后会着重讲解它们，但首先构建一个Column布局窗口，如以下代码清单所示。

代码清单5-7　探究Column布局

```
var myWin = Ext.create("Ext.Window", {
    height        : 200,
    width         : 400,
    autoScroll    : true,                                    ❶ 设置可滚屏
    id            : 'myWin',
    title         : 'A Window with a Column layout',
    layout        : 'column',                                ❷ 配置布局
    defaults      : {
        frame : true
    },
    items         : [
        {
            title       : 'Col 1',
```

```
            id          : 'col1',
            columnWidth : .3                    ❸ 设置相对宽度
        },
        {
            title       : 'Col 2',
            html        : "20% relative width",
            columnWidth : .2
        },
        {
            title : 'Col 3',
            html  : "100px fixed width",
            width : 100                         ❹ 把宽度固定为100像素
        },
        {
            Title       : 'Col 4',
            frame       : true,
            html        : "50% relative width",
            columnWidth : .5                    ❺ 配置相对宽度
        }
    ]
});
myWin.show();
```

简而言之，Column布局很易用。声明子项，并指定相对宽度或绝对宽度或两者的结合，如上所示。在代码清单5-7中，把容器的`autoScroll`属性❶设为`true`，以确保当子组件的尺寸合在一起超过容器的尺寸时，滚动条能够显示出来。接下来把`layout`属性设置为`'column'`❷。然后声明4个子组件，其中第一个子组件通过`columnWidth`❸属性把它的相对`width`设为30%。把第二个子组件的相对`width`设为20%。然后混合使用固定和相对宽度，给第三个子项设置固定`width`为100像素❹。最后，设置最后一个子项的相对`width`❺为50%。渲染后的示例应该如图5-8所示。

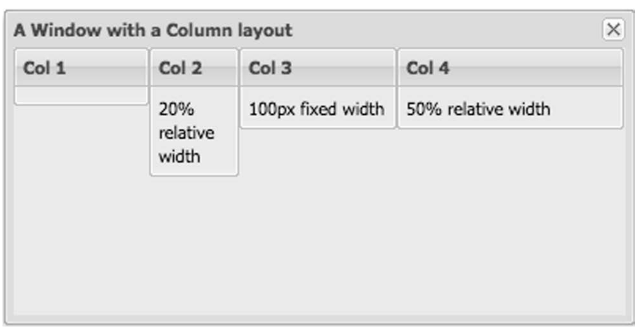


图5-8　首个Column布局，采用相对于一个固定列宽实体的相对列宽

如果把所有相对宽度相加，会发现它们的总和是100%。这是怎么回事？三个组件，占据了100%的宽度，可另外还有一个固定宽度组件？要理解其中的原因，你要深入解析一下Column布局是如何设置所有子组件的尺寸的。让我们再来动动数学脑筋。

Column布局的精髓是它的onLayout方法，后者计算容器内容体的尺寸，在这里也就是388像素。然后它遍历所有直接子项，确定分配给所有带相对宽度的子项的空间大小。

为此，它首先要用容器内容体的已知宽度，剪去所有带绝对宽度的子组件的宽度。在这个示例中，有一个绝对宽度为100像素的子项。Column布局计算出388和100的差，也就是288（像素）。

现在Column布局知道还剩下多少水平空间，它可以根据百分比来设置每个子组件的尺寸。它遍历每个子项，并基于已知可用的容器内容体水平宽度，来设定每个子项的尺寸。具体来说就是拿可用容器内容体水平宽度乘以百分比（用小数表示）。设置完成后，所有相对宽度的子组件的宽度总和就是288像素。

现在你已经了解了这种布局的宽度计算方法，让我们把注意力转移到子项的高度上去。请注意子组件的高度并不等于容器内容体的高度，这是因为Column布局并不管理子组件的高度。这给子项们带来了一个问题，它们可能会超出它们容器主体的高度。这就是为什么要把这个窗口的autoScroll设为true。你可以往'Col1'组件添加一个超大的子项来践行这一理论。在Firebug的JavaScript输入控制台里输入以下代码。确保浏览器里正在运行代码清单5-7的原始代码。

5

```
Ext.getCmp('col1').add({
    height : 250,
    title  : 'New Panel',
    frame  : true
});
```

现在你应该看到一个面板嵌在面板'Col1'中，其高度超出了窗口内容体的高度。请注意滚动条是如何出现在窗口中的。如果没有把autoScroll设为true，UI看起来会断的，其可用性可能打折扣或者有缺陷。可以垂直和水平地滚屏。之所以可以垂直滚屏，原因在于Col1的整体高度大于窗口内容体的高度。这是可以接受的。这里的问题在于水平滚屏。你可能还记得Column布局计算出只有288像素可以供带相对宽度的三列分配空间。因为垂直的滚动条现在可见了，所以可以用来显示三列的物理空间又要减去垂直滚动条的宽度。在Ext JS 4中，在往任何直接子项添加组件的时候，都会自动调用父容器的doLayout方法（在之前的版本中，必须要调用父容器的doLayout方法以保持UI美观）。

可以看到，Column布局非常适合把子组件分列进行组织。这种布局有两个限制。所有子项都是始终左对齐的，而它们的高度都不受父容器支配。Ext JS提供了HBox布局来帮助克服Column布局的局限性，并大大扩展它的功能。

5.9 HBox 和 VBox 布局

HBox布局的行为跟Column布局类似，因为它把元素分列显示，但它的灵活性可以大得多。比如说，既可以在垂直方向又可以在水平方向上改变子项的排列方式。这个布局方案另一大特点在于在需要的时候它可以让列和行伸展到父容器的对应尺寸。

让我们来深入探讨一下HBox布局，如以下代码清单所示，在其中将创建一个有三个子面板

的容器。但首先，让我们看看图5-9，了解一下要实现的效果。

代码清单5-8　HBox布局：探究包装配置

```
Ext.create("Ext.Window", {
    layout       : 'hbox',
    height       : 300,
    width        : 300,
    title        : 'A Container with an HBox layout',
    layoutConfig : {
        pack : 'start'
    },
    defaults : {
       frame : true,
       width : 75
    },
    items : [
       {
          title  : 'Panel 1',
          height : 100
       },
       {
          title  : 'Panel 2',
          height : 75,
          width  : 100
       },
       {
          title  : 'Panel 3',
          height : 200
       }
    ]
}).show();
```

❶ 把布局设置为'hbox'

❷ 指定布局配置

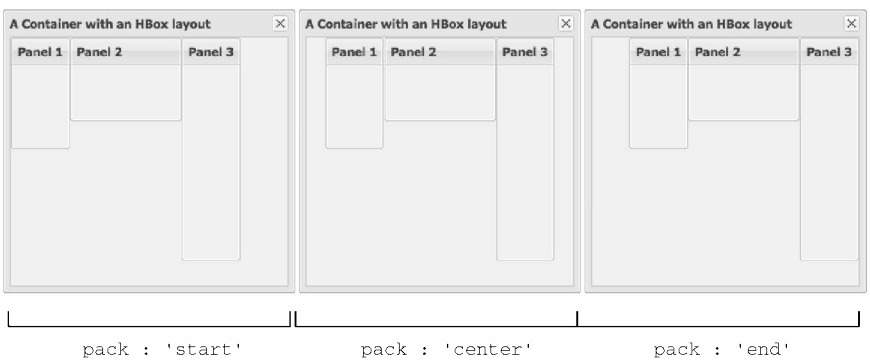


图5-9　HBox布局选项（代码清单5-8）

在代码清单5-8中，你把layout设为'hbox'❶并且指定layoutConfig❷配置对象。把三个子面板创建为不规则的形状，让你可以充分地练习不同的布局配置参数。在这些布局配置参数中可以指定两个，即pack和align，其中pack意思是“垂直对齐方式”而align意思是“水平对

齐方式”。理解这两个参数的含义之所以重要，是因为对HBox布局的近亲VBox布局来说，它们正好相反。pack参数接受三个可能的赋值：'start'、'center'和'end'。在这里我们把它们理解成左、中和右。修改代码清单5-8里的参数会导致图5-9中的一个渲染后窗口。pack属性的默认值是'start'。

align参数接受4个可能的赋值：'top'、'middle'、'stretch'和'stretchmax'。请记住在HBox布局中，align属性指定的是垂直对齐方式。

align属性的默认参数是'top'。为了改变子面板的垂直对齐方式，必须要覆盖它的默认值，在容器的layoutConfig对象中加以指定。图5-10展现了如何根据几个不同的属性组合来改变子项设置尺寸和组织排列的方式。

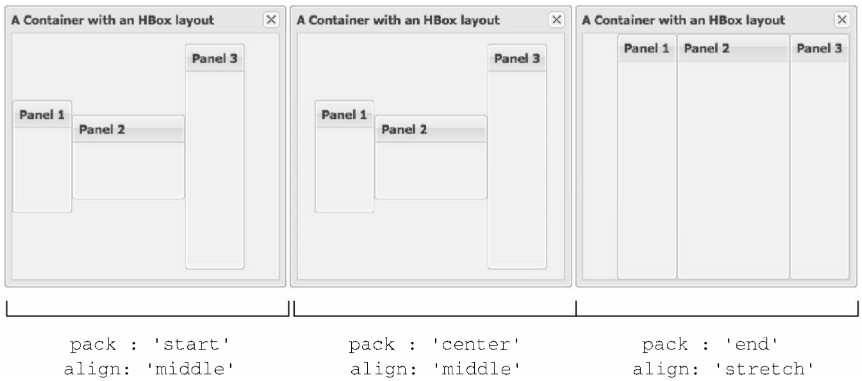


图5-10 'stretch'排列选项将总是覆盖子项指定的任何高度值

给align属性指定一个值'stretch'，也就是指示HBox布局把子项的高度调整为容器内容体的高度，这样可以克服Column布局的局限。

我们必须要探究的最后一个配置参数是flex，它跟Column布局的columnWidth参数类似，并在子项中指定。和columnWidth参数不同，flex参数被诠释为一种权重，或者说优先级，而不是列的百分比。比如说，希望能够每一列的列宽都相同。那就把每一列的flex设为相同值，那它们的宽度就一致了。如果想让两列的总列宽拓宽到父元素容器宽度的1/2，第三列拓宽为剩余的1/2，那就让前两列的flex值都等于第三列flex值的1/2。比如说：

```
defaults  : {
    frame : true,
    width : 75
},
items     : [
    {
        title : 'Panel 1',
        flex  : 1
    },
    {
        title : 'Panel 2',
        flex  : 1
```

```
        },
        {
            title : 'Panel 3',
            flex  : 2
        }
    ]
```

用VBox布局也可以垂直堆叠元素，采用的语法跟HBox布局相同。要使用VBox布局，修改代码清单5-8中的代码，把`layout`属性改成`'vbox'`，并且刷新页面。然后，可以应用之前描述的`flex`参数，让每个面板的高度都是与父容器的相对高度。我们喜欢把VBox布局看成是Auto布局的加强版。

把VBox布局和HBox布局相对照，有一个参数变化。回想一下HBox布局的`align`参数接受赋值`'top'`。而对VBox布局来说，指定的是值是`'left'`而非`'top'`。

现在你已经掌握了HBox和VBox布局，我们要改换到Table布局，你可以在上面安置子组件，比如一个传统的HTML表格。

5.10　Table 布局

Table布局可以全面控制如何清晰可见地组织排列组件。有很多人习惯于用传统方式来构建HTML表格，也就是写HTML代码。构建一个Ext JS组件的表格则不一样，因为要用一个一维数组来指定表格单元格的内容，这可能会有点让人迷惑。

我们可以肯定，完成这些练习后你将成为使用这种布局的专家。在代码清单5-9中，你将创建一个基本的3 × 3 的Table布局，如图5-11所示。

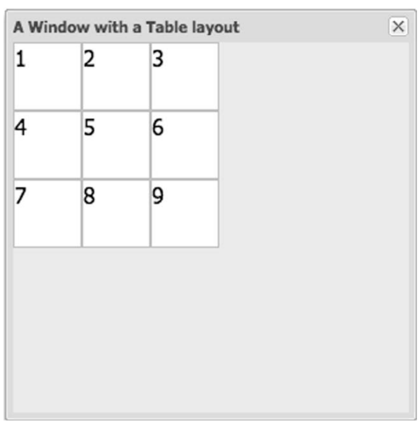


图5-11　代码清单5-9中首个Table布局实现的结果

代码清单5-9　一个基本的Table布局

```
    var myWin = Ext.create("Ext.Window", {
        height        : 300,
```

```
        width         : 300,
        border        : false,
        autoScroll    : true,
        title         : 'A Window with a Table layout',
        layout        : {                                    ❶ 指定布局为'table'
            type      : 'table',
            columns   : 3
        },                                                   ❷ 设置列数
        defaults      : {
            height    : 50,                   ❸ 配置默认尺寸
            width     : 50
        },
        items         : [
            {
                html : '1'
            },
            {
                html : '2'
            },
            {
                html : '3'
            },
            {
                html : '4'
            },
            {
                html : '5'
            },
            {
                html : '6'
            },
            {
                html : '7'
            },
            {
                html : '8'
            },
            {
                html : '9'
            }
        ]
    });
    myWin.show();
```

代码清单5-9中的代码创建了一个窗口容器，其中有九个框以3×3的排列方式堆叠，如图5-11所示。到现在这段代码大多数应该看似很熟悉，不过我们要着重说几点。其中最明显的一点应该是布局type参数❶，设为'table'。接下来，设置一个布局column属性❷，从而指定列的数量。当使用这种布局时一定要记得设置这个属性。最后，把所有子项的defaults❸属性设为50像素宽×50像素高。

经常需要让表格的某些部分跨越多行或者多列。为了实现这一点，必须要在子项上显式指定rowspan或者colspan参数。结束后，布局应该如图5-12所示。

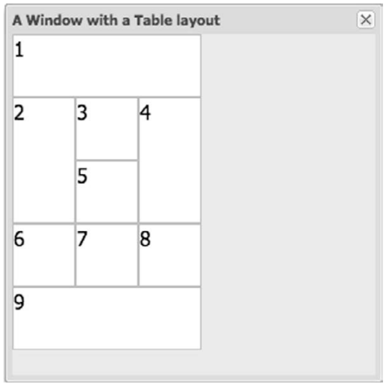


图5-12　在使用Table布局的时候，你可以为某一个特定组件指定rowspan和colspan，这可以令它在表格中包含不止一个单元格

我们来修改表格，让子项可以跨越多行或者多列，如以下代码清单所示。

代码清单5-10　探究rowspan和colspan

```
items : [
    {
        html    : '1',
        colspan : 3,      ① 设置跨列数为3，宽度为150像素
        width   : 150
    },
    {
        html    : '2',
        rowspan : 2,      ② 设置跨行数为2，高度为100像素
        height  : 100
    },
    {
        html : '3'
    },
    {
        html    : '4',
        rowspan : 2,      ③ 设置跨行数为2，高度为100像素
        height  : 100
    },
    {
        html : '5'
    },
    {
        html : '6'
    },
    {
        html : '7'
    },
    {
        html : '8'
    },
    {
```

```
        html    : '9',
        colspan : 3,
        width   : 150
    }
]
```

❹ 设置跨列数为3，宽度为150像素

在代码清单5-10中，重用了现有的代码清单5-9中的Container代码，并替换了其子items数组。把第一个面板❶的colspan属性设为3，并且手动设定它的宽度，以符合已知的表格整体宽度，也就是150像素。请记住有三列默认尺寸为50像素×50像素的子容器。接下来，把第二个子项❷的rowspan属性设为2，把它的高度设为两行的总高度，也就是100像素。对面板4也进行同样的操作❸。最后的改动涉及面板9，它的属性设置跟面板1一模一样❹。改动后渲染的表格应该如图5-12所示。

在使用Table布局时，要记住几件事。首先，要确定会用到的列的数量，并在布局的column配置属性中指定。其次，如果要让组件跨越多行和/或多列，就一定要相应地设置它们的尺寸，否则分布在表格中的组件就会无法正确地对齐。Table布局通用性极强，可以用来创建你所想象出的任何一种基于框的布局，它的主要局限是它没有父子尺寸管理功能。

5

我们的Ext JS布局之旅走到了最后一站，那就是曾经很受欢迎的Border布局，它可以把任何容器划分成5个可收缩的区域，管理它们子项的尺寸。

5.11　Border 布局

Border布局2006年首次推出，那时Ext还只不过是YUI类库的一个扩展。打那之后它已经成长为一个具有极高灵活度和易用性的布局，让容器可以完全控制其子项或者区域。Border布局被公认为是一种把复杂应用划分成多个可管理区域的简单方法并得到广泛应用。这些区域被恰当地根据极地坐标命名为北、南、东、西和中。图5-13展现了一个用Ext JS开发工具包实现的Border布局。

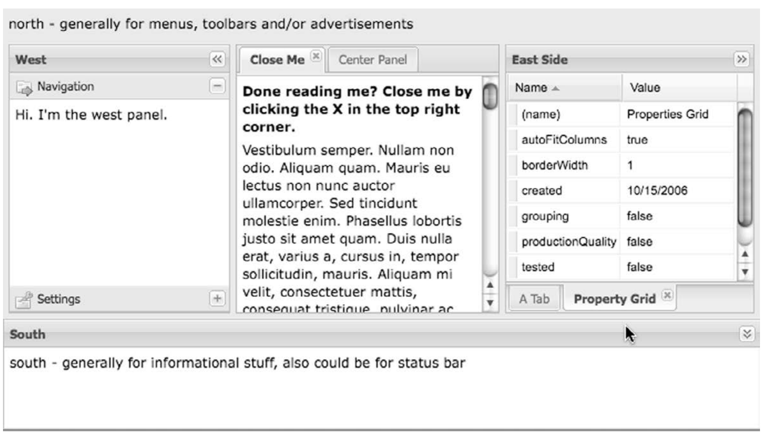


图5-13　Border布局吸引了很多Ext JS框架的新开发人员，它在很多应用中得到使用，把屏幕划分为以任务细分的功能区域

根据提供的配置选项不同，用户可以对区域进行尺寸调整或者展开折叠。还有一些配置选项被用于限制区域的尺寸调整，或者防止区域被同时调整尺寸。

为了探究Border布局，我们要用`Viewport`类，如代码清单5-11所示，这样就更可以更容易看到这项练习的最终结果。

代码清单5-11　尝试Border布局

```
Ext.create('Ext.Viewport', {
    layout    : 'border',
    defaults : {                          ❶ 分割区域，允许调整尺寸
        frame : true,
        split : true
    },
    items : [
        {                                 ❷ 添加北部区域
          title       : 'North Panel',
          region      : 'north',
          height      : 100,
          minHeight   : 100,
          maxHeight   : 150,
          collapsible : true
        },
        {                                 ❸ 设置可调整尺寸的南部区域
          title       : 'South Panel',
          region      : 'south',
          height      : 75,
          split       : false,
          margins     : {
             top      : 5
          }
        },
        {                                 ❹ 配置东部区域
          title       : 'East Panel',
          region      : 'east',
          width       : 100,
          minWidth    : 75,
          maxWidth    : 150,
          collapsible : true
        },
        {                                 ❺ 添加西部区域
          title        : 'West Panel',
          region       : 'west',
          collapsible  : true,
          collapseMode : 'mini',
          width        : 100
        },
        {
          title  : 'Center Panel',
          region : 'center'
        }
    ]
});
```

在代码清单5-11中，短短几行代码你就用Viewport实现了很多功能。在默认的配置对象中把layout设为'border'❶，并把split设为true。这里同时在发生很多事情,所以请随时参考图5-14，其中描绘了渲染后的代码是何模样。

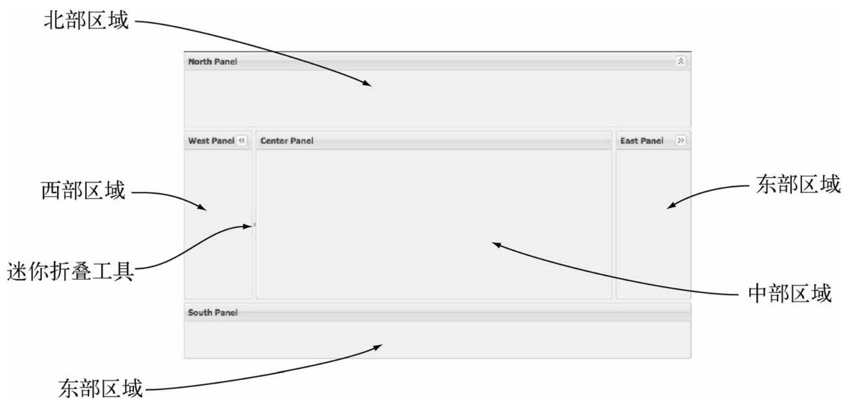


图5-14　Border布局的多功能性和易用性令它成为基于Ext JS的富互联网使用应用中最广泛的布局之一

然后，开始实例化子项，它们都拥有Border布局区域特有的参数。为了方便查看众多参数，要让每个区域的行为不同于其他区域（见图5-14）。

对第一个子项❷，把region属性设为'north'以确保它位于Border布局的顶端。对于带框组件特有的参数height，以及区域特有参数minHeight和maxHeight，要来玩点花样。指定height为100，也就是指示该区域把面板的初始高度渲染为100像素。minHeight参数指示相应的区域不允许把分隔条拖到使北部区域满足最小高度100像素的坐标外部。maxHeight参数也是同样的道理，只不过它是用于拓展区域的高度。还要把面板特有参数collapsible指定为true，指示该区域允许其折叠到仅仅30像素高。

定义南部区域，也是视口的第二个子项❸，设置一些配置项以防止它调整尺寸，但是保持布局的区域之间5像素的分隔条。通过把split设为false，设置该区域不允许调整尺寸。这样做也是在配置该区域忽略5像素的分隔条，这会使得该布局在外观上有些不完整。为获得修饰性的分隔条，要用到一个区域特有参数margin，通过它来指定南部区域与其上方的其他区域之间有一个5像素宽的缓冲区。关于这一点要提醒一句：虽然现在布局看起来完整了，但最终用户或许还会试图调整尺寸，可能令他们徒增烦恼。

第三个子项❹被定义为东部区域。这个区域的配置基本上跟北部面板一样，但它对尺寸调整的约束要更灵活一点。北部区域刚开始的时候被设为其最小尺寸，而东部区域的初始尺寸介于它的minWidth和maxWidth之间。像这样指定尺寸参数可以让UI以默认或者指定的尺寸显示一个区域，同时也使得面板可以调整到超出其原始尺寸。

西部区域❺有一个特殊的区域指定参数collapseMode设为字符串'mini'。这样设置参数指示Ext JS把一个面板折叠到只有5像素，从而为中心区域提供更多的可视空间。图5-15展现了区

域在折叠后可以多小。通过允许split参数保持true（还记得defaults对象吗）并且不指定最小或者最大尺寸参数，西部区域可以在浏览器允许的范围内随意调整尺寸。

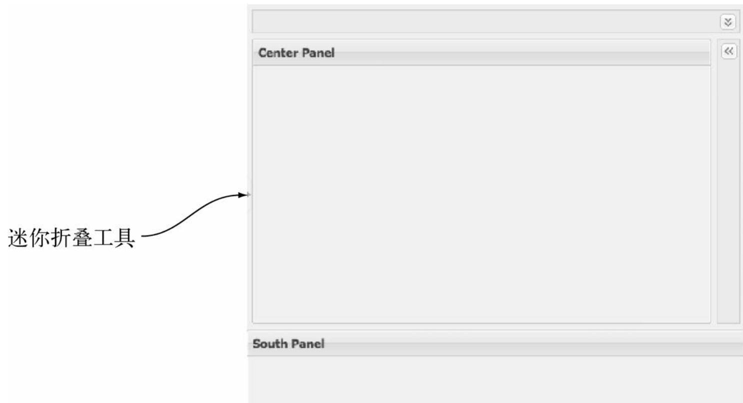


图5-15　Border布局，其中两个区域（北部和东部区域）在常规模式下折叠，而西部的面板在微缩模式下折叠

最后一个区域是中心区域，它是Border布局唯一必需的区域。虽然中心区域看起来空空的，可其实它很特别。中心区域通常开发人员放置富网络应用UI组件的背景，它的尺寸取决于其姊妹区域的尺寸。

Border布局虽然有很多优点，但也有一个很大的缺点，那就是一个子项一旦在一个区域中定义或者创建，就无法更改。而解决方法非常简单。对每一个想要更换组件的区域，都指定一个容器作为区域。让我们尝试一下，替换掉代码清单5-11中的中心区域代码：

```
{
    xtype  : 'container',
    region : 'center',
    layout : 'fit',
    id     : 'centerRegion',
    items  : {
        title : 'Center Region',
        id    : 'centerPanel',
        html  : 'I am disposable',
        frame : true
    }
}
```

请记住视口只能创建一次，所以必须要刷新一次示例代码所在的页面。刷新过的视口应该看起来跟图5-15几乎一模一样，唯一不同的是现在中心区域有HTML显示它是可自由处置的。在上面这个示例中，你用一个值为'fit'的layout属性和一个可以用在Firebug JavaScript控制台中的id定义了容器的XType。

回想我们之前关于在容器上添加和删除子组件的讨论和练习，还记得怎么根据组件的id获得该组件的引用并删除一个子项吗？如果能记得，那好极了！如果记不得，那我们已经替你完成了！

不过一定要复习之前的章节，因为它们对于管理Ext JS UI非常重要。我们来尝试一下替换中心区域的子组件，如以下代码清单所示：

代码清单5-12　替换中心区域的一个组件

```
var centerPanel  = Ext.getCmp('centerPanel'),
    centerRegion = Ext.getCmp('centerRegion');

centerRegion.remove(centerPanel, true);

centerRegion.add({
    xtype       : 'form',
    frame       : true,
    bodyStyle   : 'padding: 5px',
    defaultType : 'field',
    title       : 'Please enter some information',
    defaults    : {
        anchor : '-10'
    },
    items       : [
        {
            fieldLabel : 'First Name'
        },
        {
            fieldLabel : 'Last Name'
        },
        {
            xtype      : 'textarea',
            fieldLabel : 'Bio'
        }
    ]
});
```

代码清单5-12用到了你目前学到的所有与组件、容器和布局相关的知识，给你提供了较大的灵活性，以相对简单的方法把中心区域的子项，也就是一个面板替换为一个表单面板。你可以在任何区域内使用这种办法随意替换各项。

5.12　小结

本章探究了为数众多且用途广泛的Ext JS布局方案，带你了解了各种布局的强项、弱项和缺陷。请记住，虽然很多布局可以实现相似的功能，但它们在UI里各司其职。到底用哪种布局来显示组件或许并不是那么一目了然，如果你是UI设计新手的话是需要一些练习的。

如果在看完后对于本章的内容感觉并非100%有把握，那么我们建议你往下看，过一段时间以后再来复习一下，这些内容需要花点儿时间消化吸收；最好是在你开始学习本书第三部分时。

现在我们已经讲解了很多核心主题，现在请系好安全带，因为下面的内容更精彩。接下来，你要学习更多关于Ext JS UI部件的内容，首先是表单。

第 6 章

Ext JS中的表单

6

本章内容

- ❑ 探究表单面板输入框
- ❑ 创建自定义组合框模板
- ❑ 创建一个复杂的布局表单面板

前文介绍了如何用Ext JS框架中的多种布局管理器组织UI部件。本章开始，我们将切入实例化和管理Ext JS表单元素的内容。毕竟，没有用户输入怎么能算是应用呢？

所以也难怪开发和设计表单是网页开发人员的一项常规任务。管理表单校验是几年前JavaScript的主要用途。Ext JS超越了典型的表单校验，基于基本的HTML输入框构建而成，既为开发人员增加了特色功能又提升了用户体验。举例来说，如果用户被要求往一个表单中输入HTML。使用原始的多行文本框时，用户必须手工输入HTML内容。在EXT JS HTML编辑器中则无需如此，你可以获得一个完全“所见即所得”的输入框，使用户可以轻松地输入内容和操作格式丰富的HTML。

本章，我们将探讨表单面板，介绍Ext JS的众多表单输入类。你还将了解如何基于布局和容器模型构建复杂的表单，并用它通过Ajax提交和读取数据。

因为输入框要谈的东西有很多，本章将遵循类似食谱的风格为你依次讲解很多Ext JS输入框，比如一般的单行文本输入框、多行文本框和数字输入框。我们要来详细探究组合框，那是一种结合了简单的单行文本框与自定义下拉框的输入框，可以算得上是Ext JS框架中实现起来最复杂的输入框。当你扎实地掌握了这些输入框的相关知识后，要把它们融会贯通，我们将一起实现并讨论`FormPanel`类，让你详细了解如何保存和读取数据。

6.1　基本输入框

Ext JS表单输入控件及其子类给现有的HTML输入框增加了诸如基本校验，自定义认证方法，自动调整大小和键盘过滤之类的功能。要使用诸如键盘过滤器（掩码）和自动字符剥离之类的更强大功能，你就需要了解正则表达式。

进一步了解如何在JavaScript中使用正则表达式

如果不熟悉正则表达式，网上有大量的相关信息供你参考。我们最喜欢的学习该主题的网站之一是：www.regularexpressions.info/javascript.html。

6.1.1 输入框和校验

我们要一次性地探讨输入框的好几个功能。请务必耐心，因为某些示例代码可能会很长。

输入框是作为表单面板的子元素构建的，以便于跟踪显示方面的问题。首先你要创建items数组，其中将包含不同单行文本框的XType定义，如以下代码清单所示。

代码清单6-1 文本输入框

```
Ext.QuickTips.init();
 var fpItems =[
    {
        fieldLabel : 'Alpha only',
        allowBlank : false,
        emptyText  : 'This field is empty!',     ❶ 指定空输入框
        maskRe     : /[a-z]/i ,                  ❷ 指定唯字母字符
        msgTarget  : 'side'
    },
    {
        fieldLabel : 'Simple 3 to 7 Chars',
        allowBlank : false,
        msgTarget  : 'under',                    ❸ 容许的最小/最大字符数
        minLength  : 3,
        maxLength  : 7
    },
    {
        fieldLabel   : 'Special Chars Only',
        msgTarget    : 'qtip',
        stripCharsRe : /[a-zA-Z0-9]/ig           ❹ 只容许特殊字符
    },
    {
        fieldLabel : 'Web Only with VType',      ❺ 使用url VType
        vtype      : 'url',
        msgTarget  : 'side'
    }
];
```

在代码清单6-1中，你必须要从很多角度入手展示简单的单行文本框的功能。fpItems数组中创建的是4个单行文本框。每个子元素所拥有的冗余属性之一是fieldLabel，也就是显示在field元素对应的label元素中的文本。

对于第一个子元素，你要指定allowBlank为false确保该输入框不能为空，从而也确保使用到Ext JS的一项基本输入框校验功能。你还给emptyText赋了一个字符串值❶，显示帮助文本，它可以被用作默认值。有一点很重要值得注意，那就是它可以在表单提交中作为该输入框的值提

交。接下来设置正则表达式掩码maskRe❷，使之过滤掉那些非字母字符的按键。第二个单行文本框被设计为不能为空，而且必须输入3~7个字符才能通过校验。你可以通过设置minLength❸和maxLength参数来实现。第三个单行文本框可以为空，但它有自动字数字符剥离功能。你可以给stripCharsRe属性❹指定一条有效的正则表达式来实现自动剥离。对于最后一个子元素❺，你使用VType url来检查输入的值是否是一个URL。以下代码清单将创建一个表单面板来渲染输入框。

代码清单6-2　为单行文本框构建表单面板

```
var fp = Ext.create('Ext.form.Panel', {
    renderTo      : Ext.getBody(),
    width         : 400,
    height        : 240,
    title         : 'Exercising textfields',
    frame         : true,
    bodyStyle     : 'padding: 6px',
    labelWidth    : 126,
    defaultType   : 'textfield',
    defaults      : {
       msgTarget  : 'side',
       anchor     : '-20'
    },
    items         : fpItems
});
```

❶ 给文本输入框设置默认XType

❷ 设置校验信息目标

代码清单6-2中的大部分代码你都应该很熟悉。不过让我们来复习几个与组件模型相关的关键项。把defaultType属性❶设为'textfield'，以此来覆盖默认的组件Xtype，而如果你还记得，这么做可以确保你的对象融入单行文本框。你还设置了一些默认值❷，这可以确保报错信息目标位于输入框右侧，此外设置anchor属性。最后把表单面板的items配置设置成之前创建的ftpItems变量，其中包含4个单行文本框。渲染后的表单面板应该如图6-1所示。



图6-1　表单面板的渲染结果，其中包含4个单行文本框

请注意在图6-1中，文本输入框右侧有一点多余的空间。这是因为必须确保校验报错信息显示在单行文本框的右侧。这就是把表单面板定义中默认对象的msgTarget设为'side'的原因。可以执行校验的方式有两种：聚焦或者散焦（使之失去焦点）一个输入框，或者执行一个全表单的isValid方法调用，fp.getForm().isValid()。图6-2展现了校验发生后输入框都

是什么样。

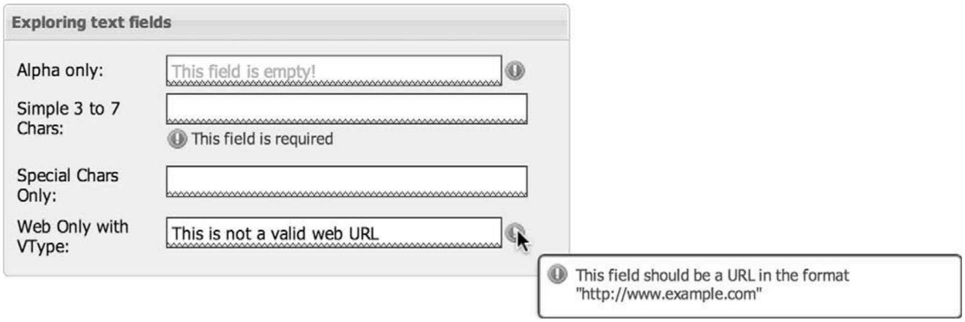


图6-2 校验错误信息

每个输入框都可以有自己的msgTarget特性，它可能是以下5种属性之一。

- qtip：触发mouseover事件会显示一个Ext JS小技巧。
- title：在默认浏览器标题区显示报错信息。
- under：把报错信息显示在输入框之下。
- side：在输入框的右侧渲染一个惊叹号图标。
- [elementid]：把报错信息的文本作为目标元素的innerHTML添加。

请注意只有在输入框位于一个输入框容器（通常是表单面板）内部时，msgTarget特性才影响报错信息的显示方式。如果单行文本框被渲染成页面上某处的某个任意元素（使用renderTo或者applyTo），那msgTarget将只被设置为title。我们建议花点时间试用不同的msgTarget值，这样你在构建首个实用表单的时候可以更好地了解它们的工作原理。让我们来看看如何使用单行文本框创建密码和文件上传框。

6.1.2 密码和文件选择框

为创建一个密码输入框，你要选择password输入类型，而对于文件输入框，要把xtype设为'filefield'。

在Ext JS中要生成这些输入框，就输入以下代码：

```
var fpItems =[
   {
      fieldLabel : 'Password',
      allowBlank : false,
      inputType      : 'password',
   },
   {
      fieldLabel : 'File',
      allowBlank : false,
      xtype      : 'filefield'
   }
];
```

图6-3展现了表单面板中密码和单行文本框的渲染版本。

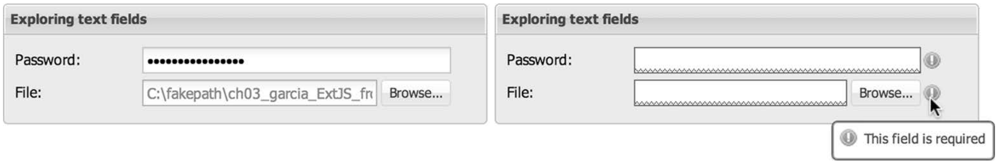


图6-3　已填写数据的密码和文件上传框（左）和一个侧面校验报错图标的示例

我们已经介绍了很多关于文本输入框、输入框校验和密码及文件上传输入框的内容，现在来看看其他输入框。

6.1.3　构建多行文本框

TextArea类扩展了TextField类。多行文本框是一个多行的输入框。构建一个多行文本框与构建一个单行文本框类似，只是必须要考虑到组件的高度。以下是一个拥有固定高度和相对宽度的多行文本框示例：

```
{
    xtype      : 'textarea',
    fieldLabel : 'My TextArea',
    name       : 'myTextArea',
    anchor     : '100%',
    height     : 100
}
```

就是这么简单。我们来快速了解一下如何使用数字输入框。

6.1.4　便利的数字输入框

有时候开发需求要求你放置一个只允许输入数字的输入框。这你可以用单行文本框来实现，而且你要施加自己的校验，但干嘛要多此一举呢？数字输入框基本上可以替你完成对证书和浮点数的所有校验。让我们来创建一个数字输入框，它接受精确到千分之一的浮点数，并且只允许输入特定值：

```
{
    xtype            : 'numberfield',
    fieldLabel       : 'Numbers only',
    allowBlank       : false,
    emptyText        : 'This field is empty!',
    decimalPrecision : 3,
    minValue         : 0.001,
    maxValue         : 2
}
```

为了把需求应用于这个数字输入框，你要指定decimalPrecision、minValue和maxValue

属性。这样可以确保任何小数点后多于3位的浮点数都会被四舍五入。同样，设置minValue和maxValue属性是为了确保输入的有效数字有效范围为0.001 ~ 2。任何在此范围以外的数字都被视为无效，Ext JS会对其进行无效标识。数字输入框渲染后外观上看跟单行文本框一样，只是增加了几个触发器（按钮），让用户可以通过鼠标点击来增大或者减少数字。另外还有几个属性可以帮助你对数字输入框进行配置。

我们已经看过了文本输入框，多行文本框和数字输入框，现在再让我们来看看它们的远亲：组合框。

6.2　用组合框实现提前键入

命名恰如其分的组合框就像是多功能的文本输入框。它融合了一个通用文本输入框和一个通用下拉框，形成了一个使用灵活，可配置性极强的组合输入框。组合框可以实现在文本输入区自动填充文本（也就是所谓的提前键入），而且它可以连接远程数据存储，与服务器端配合过滤结果。如果组合框执行的是针对一个大型数据集的远程请求，那你可以通过设置pageSize属性来实现结果分页。图6-4展现了一个远程读取和分页组合框的详细结构。

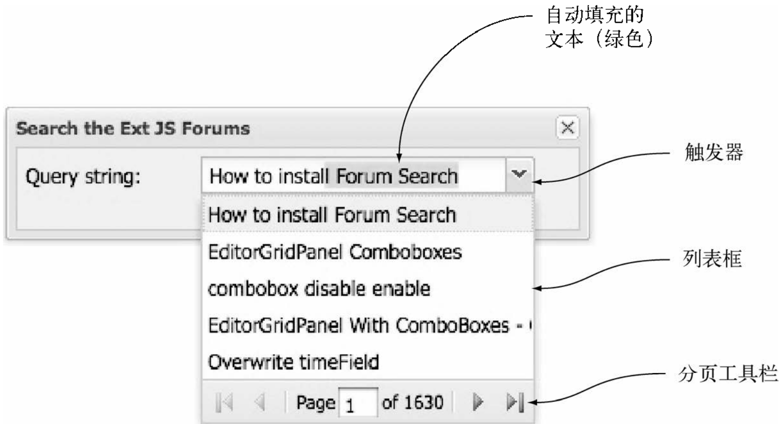


图6-4　一个带提前键入功能的远程读取和分页组合框的示例UI

在我们学习运用组合框之前，先来看看怎么构建组合框。因为你已经熟悉了如何布置子元素，这是一个把新学到的知识投入实践的大好机会。所以从现在起，当我们讨论不包含子元素的项，比如输入框的时候，我们会让你自己来构建一个容器。你可以使用代码清单6-2中的表单面板。

6.2.1　构建一个本地组合框

创建一个单行文本框相比构建一个组合框来说很简单。这是因为组合框是对于一个叫data Store的类有直接的依赖关系，它是Ext JS框架中管理数据的主要工具。本章我们将简单介绍这

个支撑类，在第7章我们将深入探讨它的细节。在下面的代码清单中，你将使用一个XType配置对象来构建第一个组合框。

代码清单6-3　构建第一个组合框

```
var mySimpleStore = ({
     type   : 'array',
     fields : ['name'],
     data   : [
        ['Jack Slocum'],
        ['Abe Elias'],
        ['Aaron Conran'],
        ['Evan Trimboli']
     ]
});
var combo = {
    xtype        : 'combo',
    fieldLabel   : 'Select a name',
    store        : mySimpleStore,
    displayField : 'name',
    typeAhead    : true,
    mode         : 'local'
};
```

❶ 构建数组数据存储

❷ 在组合框中指定数据存储

❸ 设置显示文本框

❹ 把组合框设为本地模式

代码清单6-3构建了一个读取数组数据的简单数据存储，也就是所谓的*数组数据存储*❶（它是`Ext.data.Store`类的一个预配置扩展），让你可以方便地创建一个能消化数组数据的数据存储。你填充可消耗的数组数据，并把它设置为配置对象的`data`属性。接下来把`fields`属性指定为一个数据点数组，数据存储将利用它来读取和组织记录。因为每个数组只有一个数据点，所以只指定单个数据点，并把它命名为`'name'`。而且，我们将在第7章深入探讨数据存储的细节，而你也将学到从数据记录到连接代理的全部相关知识。

你把组合框指定为一个简单的POJSO（普通老式JavaScript对象），把`xtype`属性设为`'combo'`以确保它的父容器调用正确的类。你把之前创建的简单数据存储的引用指定为`store`属性❷。还记得给数据存储设置的`fields`属性吗？`displayField`❸与组合框正在使用的数据存储对应的输入框直接相关联。因为有单个输入框，你会用那个单独的输入框，也就是`'name'`，指定`displayField`。最后把`mode`❹设为`'local'`，这可以确保数据存储不会尝试远程读取数据。正确设置这个属性很重要，因为`mode`的默认值是`'remote'`，这可以确保所有数据都通过远程请求读取。如果忘了把它设为`'local'`会导致一些问题。图6-5展现了组合框渲染后是什么样。



图6-5　代码清单6-3中在窗口内部渲染的组合框

为了探索过滤和提前键入功能，你可以立刻开始在文本输入框内输入。现在记录集里只包含4条记录，不过你可以一窥它的运行原理了。在单行文本框里输入一个或多个字母会触发提前键入。作为尝试，键入A，你会看到输入框里被自动填入了'Abe Elias'，这是列表预先选中的记录。同样的，输入'Jac'会导致'Jack Slocum'被自动填入，而它对应的记录被预先选中。就这样，一个本地组合框构建好办法。

如果有最低数量的静态数据，使用本地组合框很棒。不过它既有优点，又有缺点。它的主要优点是数据不一定要远程读取。而当有极多的数据需要解析的时候，这反而成了一个巨大的缺点，这会导致UI速度减慢、停顿，甚至戛然而止，弹出让人不乐见到的"这条脚本运行时间过长"报错框。这时候就用到远程读取的组合框了。

6.2.2 实现一个远程组合框

使用一个远程组合框比实现一个静态组合框要复杂一点。这是因为有服务器端的代码要处理，其中会引入某种类型的服务器端数据存储，比如数据库。为了让你集中精力尝试组合框，我们使用预先构建的PHP代码（http://extjsinaction.com/dataQuery.php），其中包含随机生成的名称和地址。现在让我们来实现远程组合框，如以下代码清单所示。

6

代码清单6-4 实现一个远程读取组合框

```
var remoteJsonStore = Ext.create(Ext.data.JsonStore, {
    storeId : 'people',
    fields  : [
        'fullName',
        'id'
    ],
    proxy   : {
        type   : 'jsonp',
        url    : 'http://extjsinaction.com/dataQuery.php',
        reader : {
            type          : 'json',
            root          : 'records',          <-- ❶ 指定根属性
            totalProperty : 'totalCount'
        }
    }
});

var combo = {
    xtype          : 'combo',
    queryMode      : 'remote',
    fieldLabel     : 'Search by name',
    width          : 320,
    forceSelection : true,
    displayField   : 'fullName',
    valueField     : 'id',
    minChars       : 1,                         <-- ❷ 配置自动填充阈值
    triggerAction  : 'all',
    store          : remoteJsonStore
};
```

在代码清单6-4中，你把数据存储的类型改为Ext.data.Store类的一个预配置扩展JsonStore❶，以便方便地创建可以使用JSON数据的数据存储。然后指定fields，它现在是一个数据，包含单个对象'fullname'。你还针对每条记录的ID创建了一个映射，将用于数据提交。最后为数据存储指定了一个proxy属性，在其中创建一个JsonP代理的新实例，这个工具被用于跨领域请求数据。你指示JsonP代理通过url属性从一个特定的URL读取数据。

在创建组合框的时候把forceSelection设为true，这有助于远程过滤（在这里，也包括提前键入），但这也阻止用户任意输入数据。接下来把displayField设为'fullName'，这会在单行文本框里显示全名的数据点，而且你把valueField设为'id'，可确保在请求提交组合框数据的同时，使用ID来发送数据。hiddenName属性经常被人忽视，但它很重要。因为你要显示人名但却提交ID，所以你需要借助DOM中的一个元素来存储该ID值。

minChars属性❷定义在组合框执行数据存储读取之前需要输入单行文本框的最小字符数，你可以覆盖它的默认值4。最后把triggerAction设为'all'，这指示组合框为所有数据执行一次数据存储读取查询。图6-6展现了新构建的组合框示例。



图6-6　代码清单6-4中的远程读取组合框

试验一下渲染后的结果，你会看到使用远程过滤对用户来说是多有趣的一件事。我们来看看从服务器传回的数据是如何被格式化的（图6-7）。

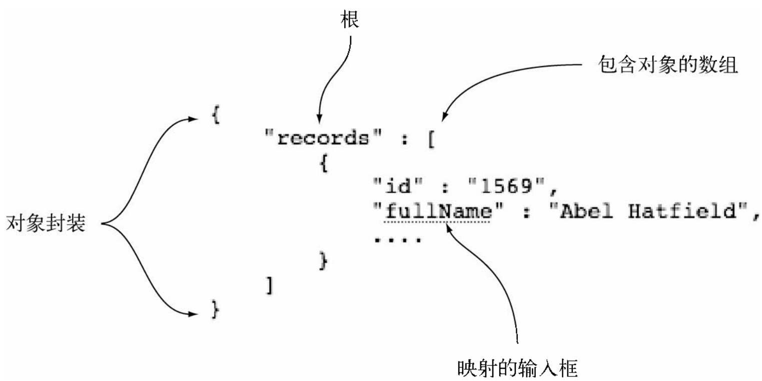


图6-7　服务器生成的JSON片段的分解图

研究这一小段生成的JSON代码，你可以看到远程的组合框JSON数据存储中指定的根，以及映射到的fullName输入框。根包含了一个对象数组，数据存储会对它加以转化。然后数据存储

将移除所有映射为"fields"的属性。因为映射了id和fullName，那些输入框将被数据存储吸收。所有其他属性都将被忽略。

在实现服务器端代码时遵循图6-7中的格式，这有助于确保你的JSON格式正确。如果你不确定，可以使用在线工具（http://jsonlint.com），粘贴自己的JSON，对其进行解析和校验。

在查看代码清单6-4中示例代码的执行结果时，你可能会注意到当点击触发器时，UI的旋转进度条会停顿一小会儿。这是因为数据库里的所有2000条记录正被发送给浏览器并进行解析，此时发生DOM操作清空列表框并创建一个节点。对于这个大数据集而言，数据的转移和解析速度相对是快的。DOM操作是JavaScript慢下来的主要原因之一，也是为什么旋转进度条动画会停止。插入2000个DOM元素所需的资源数量巨大，乃至浏览器不得不中断所有动画，把注意力集中在手头的任务上，更别提一下向用户倾泻这么多记录所可能带来的可用性问题了。为了缓解这些问题，你应该启用分页功能。

而要使用分页，服务器端代码必须要意识到这些改动，这是这一转化过程中的最大难点。幸运的是，你使用的PHP代码中已经有必要的代码来适应即将做出的改动。你要做的第一个改动是给JSON数据存储添加以下属性：


```
totalProperty :'totalCount'
```

接下来你要在组合框中启用分页功能，具体的方法是添加一个pageSize属性:

```
pageSize :20
```

这就行了！Ext JS现在准备好在组合框中启用分页功能了。刷新浏览器的代码，然后或者点击触发器或者在文本输入框里输入几个字符，你就会看到所做改动的结果，如图6-8所示。

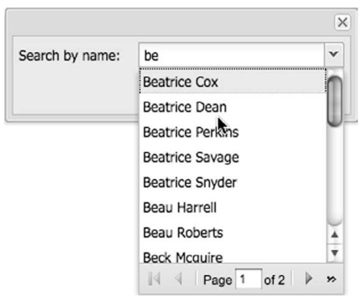


图6-8　往远程组合框中添加分页功能

目前，我们已经谈久了组合框的UI，并实现了Array和JSON数据存储的本地和远程版本。虽然我们已经介绍了组合框的很多方面，目前还只不过把它用作一个加强版的下拉框，我们还没有讨论过如何自定义结果数据的外观。为了展示为什么要改动一些东西，比如内模板，我们必须先来快速了解一下组合框的内部机制。

6.2.3　解构组合框

在组合框的内核中有两个辅助类。我们已经接触过了数据存储，它提供了数据读取和加载功能，但我们还没有讨论数据视图，这个组件负责在列表框里显示结果数据，以及提供必需的事件以允许用户选择数据。数据视图通过注册到诸如'beforeload'、'datachanged'和'clear'的事件来绑定数据存储。它们使用XTemplate，XTemplate提供DOM操作，以基于你提供的HTML模板来生成HTML。你已经简单看过了一个组合框的组件，现在可以继续创建自定义组合框了。

6.2.4　自定义组合框

在组合框中启用分页功能后，你只能看到名字。可如果想看到完整的地址以及所要搜索的名字呢？数据存储需要知道你想显示的项。修改代码清单6-4：需要添加对街道、城市、州和邮编的映射。我们会等到你完成修改。

准备好了吗？在创建一个模板之前，你必须创建一些稍后会用得着的CSS：

```
.combo-name {
    font-weight:      bold;
    font-size:        11px;
    background-color: #FFFF99;
}
.combo-full-address {
    font-size: 11px;
    color:     #666666;
}
```

这段CSS代码为模板中的每个div创建了一个类。现在需要创建一个新的模板，让列表框可以显示你想显示的任何数据。

输入以下代码以创建组合框：

```
var combo = Ext.create('Ext.form.field.ComboBox', {
    fieldLabel     : 'Search by name',
    forceSelection : true,
    displayField   : 'fullName',
    loadingText    : 'Querying....',
    pageSize       : 20,
    width          : 320,
    minChars       : 1,
    valueField     : 'id',
    triggerAction  : 'all',
    store          : remoteJsonStore,
    listConfig     : {
        getInnerTpl : function() {
            return ' <div data-qtip="{fullName}">' +
                '<div class="combo-name">{fullName}</div>' +
                '<div class="combo-full-address"> {street} </div>' +
                '<div class="combo-full-address">' +
                    '{city} {state} {zip}</div>' +
```

```
                '</div>';
            }
        }
    });
```

在这里我们不会太多深入探讨XTemplate，因为我们在第2章已经讲过了。很重要的一点是要注意，你通过覆盖`listConfig`来指定`getInnrTpl`，这是一个返回字符串的函数。这个`getInnerTpl`函数将被`ListBox`类调用，而字符串将被用于创建一个XTemplate实例。

现在可以对所做改动进行检验了。如果你做得对，结果应该与图6-9展示的内容类似。

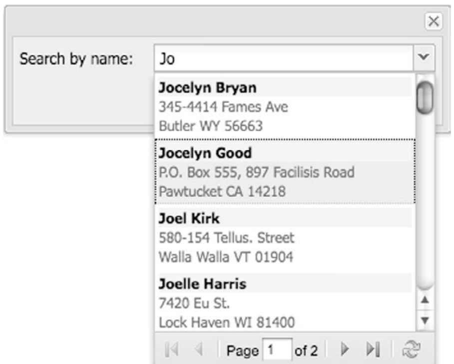


图6-9 你的自定义组合框

自定义组合框的方式还只是冰山一角！因为你对于列表框如何渲染有着全盘的控制，所以甚至可以在列表框里加入图像或者QuickTip（快速提示）。

本节介绍了如何创建一个本地和一个远程的组合框，阐述了`ArrayStore`和`JsonStore`数据存储类。你在实现的远程组合框中尝试了添加分页功能，结构了组合框，并且自定义了列表框。现在让我们再来看时间框。

6.3 时间输入框

`TimeField`也是一个便利类，让你可以方便地往表单上添加一个时间选择输入框。为了构建一个一般时间输入框，你可以创建一个`xtype`设为`'timefield'`的配置对象，并将获得一张列表，上面的可选项从中午12:00到午夜11:45。以下是实现的示例：

```
{
    xtype      : 'timefield',
    fieldLabel : 'Please select time',
    anchor     : '100%'
}
```

图6-10展现了这个输入框会如何在屏幕上渲染。时间输入框是可配置的，你可以设置时间范围，间隔甚至是格式。通过添加以下属性来修改时间输入框，这些属性可以让你使用军用时间，

设置30分钟的时间增量间隔，以及把时间选择范围限定在上午9点到下午6点：

```
...
   minValue  : '09:00',
   maxValue  : '18:00',
   increment : 30,
   format    : 'H:i'
```

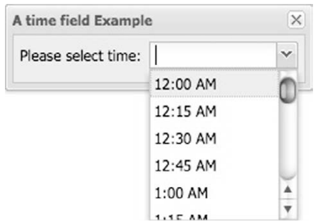


图6-10　一般时间输入框

在这个属性列表中你设置了minValue和maxValue属性，它们给时间输入框限定了时间范围。你还把increment属性设为30，把格式设为'H:i'，也即是24小时制和两位分钟数。format属性必须要能通过Date.parsetDate方法的校验。如果有意使用一种自定义格式，请查阅完整的API文档。

既然你已经看到了如何使用组合框和时间输入框，现在再来看看HTML编辑器。

6.4　HTML 编辑器

Ext JS HTML编辑器是一种所见即所得编辑器。这是一种好方法，让用户可以输入丰富HTML格式的文本，而又不用逼着他们掌握HTML和CSS。它可以让你配置工具栏上的按钮，以防止客户进行某些交互操作。接下来构建第一个HTML编辑器。

6.4.1　构建第一个HTML编辑器

构建一般的HTML编辑器很简单：

```
var htmlEditor = {
   xtype      : 'htmleditor',
   fieldLabel : 'Enter in any text',
   anchor     : '100% 100%'
}
```

渲染在一个表单上的HTML编辑器应该如图6-11所示。

我们讨论了HTML编辑器的工具栏可以如何配置以防止显示出一些工具。只要把enable<someTool>属性设置为false就可以轻松实现。比如说，如果想禁用“字体大小”和“字体选择”菜单项，你可以把一下属性设为false：

```
enableFontSize : false,
enableFont     : false
```

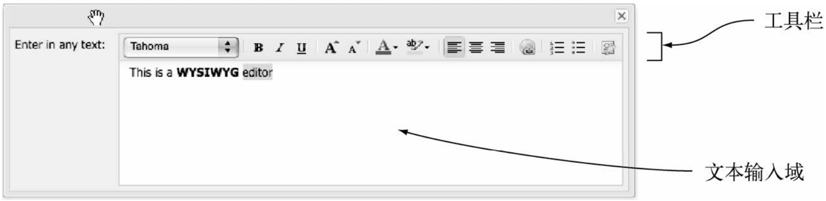


图6-11　第一个在Ext JS窗口中的HTML编辑器

就是这么简单。做出这些改动后，请刷新页面。你再也不会看到改变字体大小的文本下拉框菜单和图标。要想看到现有选项的完整列表，请务必查看API文档。HTML编辑器是一个很不错的工具，不过和很多东西一样，它也有一些局限性。

6.4.2　处理缺少校验的问题

HTML编辑器最大的一个局限性就是它没有基本的校验功能，无法把输入框标示为输入无效。在使用该输入框开发表单的时候，你必须创建自定义校验方法：一个简单的validate方法应该如下所示：

```
var htmlEditor = Ext.create('Ext.form.HtmlEditor', {
    fieldLabel : "Enter in any text",
    anchor     : '100% 100%',
    allowBlank : false,
    validate   : function () {
        var val = this.getValue();
        return (this.allowBlank || val.length > 1);
    }
});
```

虽然当消息框为空或者包含一个简单的换行元素时validate方法会返回false，但它不会把该输入框标示为“输入无效”。我们将在本章稍后讨论如何在表单提交以前检测表单的输入是否有效。现在，让我们换个话题，去看看日期输入框。

6.5　选择日期

日期输入框是一个很有意思的小表单部件，拥有很多强大的UI功能，让用户可以或者通过输入框输入日期，或者使用DatePicker部件选择日期。让我们来构建一个日期输入框：

```
var dateField = {
    xtype      : 'datefield',
    fieldLabel : 'Please select a date',
    anchor     : '100%'
}
```

是的，就是这么容易。图6-12展现了日期输入框是如何渲染的。

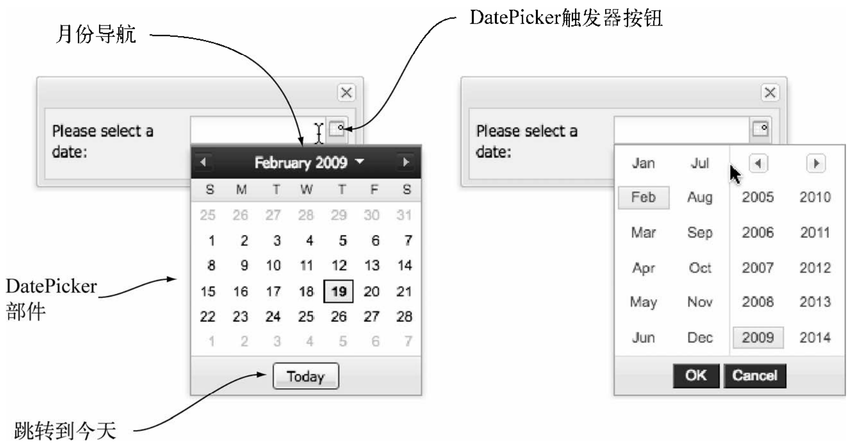


图6-12　DatePicker部件（左）暴露的日期输入栏，以及DatePicker的月和年选择工具（右）

这个部件可以通过配置防止某些日期被选中，具体方法是设定一个`date`属性，该属性是由与`format`属性相匹配的字符串组成的一个数组。`format`属性默认为*m*/*d*/*Y*，或者01/01/2001。以下为用默认格式禁用某些日期的方法：

```
["01/16/2000", "01/31/2009"] 禁用这两个日期
["01/16"]                    禁用每一年的这个日期
["01/../2009"]               禁用2009年1月的每天
["^01"]                      禁用每一个一月
```

现在你已经熟悉了日期输入框，让我们继续来探究复选框和单选按钮，学习如何使用`CheckboxGroup`和`RadioGroup`类来创建输入框集合。

6.6　复选框和单选按钮

本节的重点不仅仅是实例化复选框和单选按钮，而且还包括如何把它们并列和叠加使用。这些知识将帮助你开发允许复杂数据选择的表单。

Ext JS复选输入框基于原版的HTML复选输入框封装了Ext JS元素管理功能，同时包括布局控件。跟使用HTML复选框一样，你可以指定复选框的值，覆盖默认的布尔值。接下来要创建一些可以使用自定义值的复选框，如以下代码清单所示。

代码清单6-5　构建复选框

```
var checkboxes = [
    {
        xtype      : 'checkbox',
        fieldLabel : 'Which do you own',
        boxLabel   : 'Cat',
        inputValue : 'cat'
```

❶ 设置复选框标签文本

❷ 配置默认值

```
        },
        {
            xtype          : 'checkbox',
            fieldLabel     : ' ',
            labelSeparator : ' ',
            boxLabel       : 'Dog',
            inputValue     : 'dog'
        },
        {
            xtype          : 'checkbox',
            fieldLabel     : ' ',
            labelSeparator : ' ',
            boxLabel       : 'Fish',
            inputValue     : 'fish'
        },
        {
            Xtype          : 'checkbox',
            fieldLabel     : ' ',
            labelSeparator : ' ',
            boxLabel       : 'Bird',
            inputValue     : 'bird'
        }
    ];
```

代码清单6-5中的代码构建了四个复选框，你覆盖了其中每个节点的默认inputValue。boxLabel属性❶在输入框右侧创建了一个输入框标签，而inputValue属性❷覆盖了默认的布尔值。该段代码渲染结果的一个示例如图6-13所示。

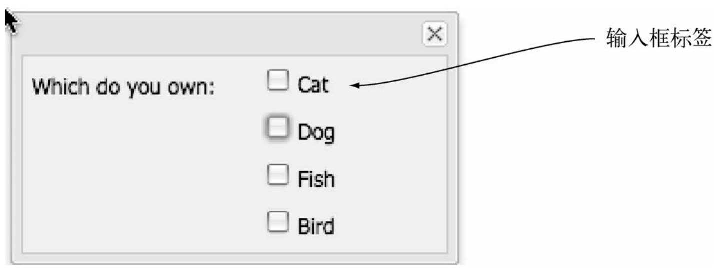


图6-13 第一个四复选框

虽然这种方法适用于很多表单，对某些大表单来说这是对屏幕空间的浪费。在下面的代码清单中，你会用复选框组来自动排列自己的复选框。

代码清单6-6 使用一个复选框组

```
var checkboxes = {
    xtype      : 'checkboxgroup',
    fieldLabel : 'Which do you own',
    anchor     : '100%',
    items      : [
       {
          boxLabel    : 'Cat',
```

```
            inputValue : 'cat'
         },
         {
            boxLabel    : 'Dog',
            inputValue : 'dog'
         },
         {
            boxLabel    : 'Fish',
            inputValue : 'fish'
         },
         {
            boxLabel    : 'Bird',
            inputValue : 'bird'
         }
      ]
   };
```

以这种方式使用复选框组会把复选框单独排列为一行，就像图6-14中显示的那样。要设定列数很简单，只需把columns属性设为想要的列数。



图6-14　复选框组的两种实现方式：单水平线（左）和双列布局（右）

具体使用哪种方式实现复选框组取决于你的需求。实现Radio和RadioGroup类的方法跟使用Checkbox和CheckboxGroup类的方法几乎一样。最大的差别在于可以给单选按钮赋予同一个名字，给它们分组，这样一次就只允许选择一个按钮。让我们来构建一个单选按钮群，如图6-15所示。

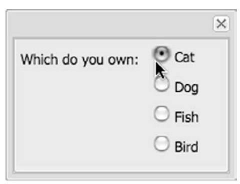


图6-15　一个单列的单选按钮

因为RadioGroup类扩展了CheckboxGroup类，所以它的实现方式是一样的，所以我们就不重复赘述了。我们已经探究了Checkbox和Radio类及其相应的Group类，现在让我们通过更深入地探究表单面板，来把它们联系在一起。你将学习如何实施全表单检查和复杂的表单布局。

6.7　表单面板

借助于Ext JS表单面板，你可以使用Ajax提交和加载数据，并且在有输入字段输入无效时向

用户提供实时反馈。因为FormPanel是Container类的一个子类，所以可以轻易地添加和删除输入字段以创建一个真正的动态表单。

> **文件上传实际并非Ajax**
>
> 在大多数浏览器中，XMLHttpRequest对象都不能提交文件数据。为了让数据提交效果看似Ajax，Ext JS使用一个iFrame来提交包含文件输入元素的表单。

一个额外的好处是表单面板可以使用其他布局或者组件，比如带Card布局的标签面板，来创建健壮的表单，相比传统布局的单页表单，对屏幕空间的占用大大减少。因为FormPanel类是Panel类的子类，所以它拥有Panel的所有功能，包括诸如工具栏之类的停靠元素。

6.7.1 检视正在构建的内容

和其他Container子类一样，FormPanel可以利用框架提供的任意布局来创建布局复杂的表单。为了配合分组输入框，表单面板有一个称为字段集的“表亲”。在构建组件以前，先来预览一下所要实现的效果（图6-16）。

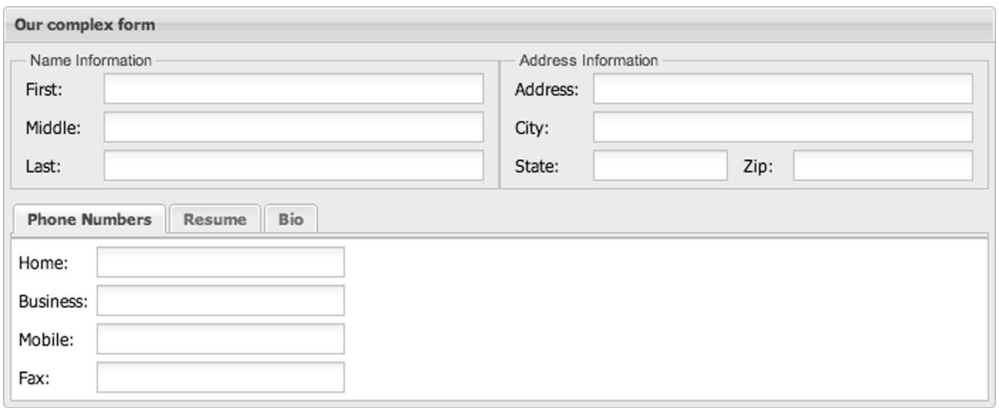


图6-16 预览你即将构建的复杂表单面板

要构建复杂表单，必须构建两个字段集：一个包含名称信息，另一个包含地址信息。除了字段集，你还要设置一个标签面板，其中可以包含一些单行文本框和两个HTML编辑器。为了完成这项任务，你要用到目前学到的所有知识，也就是说我们要来分析很多代码。

6.7.2 构建字段集

你已经知道要构建的内容了，现在先来构建字段集，其中将包含名称信息的单行文本框，如以下代码清单所示。

代码清单6-7 构建首个字段集

```
var fieldset1 = {
    xtype       : 'fieldset',                  ❶ 把xtype设为'fieldset'
    title       : 'Name',
    flex        : 1,
    border      : false,
    labelWidth  : 60,
    defaultType : 'field',
    defaults    : {
       anchor     : '-10',
       allowBlank : false
    },
    items : [
        {
          fieldLabel : 'First',
          name       : 'firstName'
        },
        {
          fieldLabel : 'Middle',
          name       : 'middle'
        },
        {
          fieldLabel : 'Last',
          name       : 'lastName'
        }
    ]
};
```

在构建首个字段集XType❶时，你可能会觉得里面的参数看起来像是面板或者容器的参数。这是因为`FieldSet`类扩展了`Container`类，并为折叠方法增加了一些功能，让你可以在表单中包含输入框。在这个例子里之所以使用字段集，是因为它可以在最顶端增加一个小标题，你通过这种方式接触到这个组件。

暂时跳过对这首个字段集的渲染，因为稍后你要在表单面板里用到它。让我们继续构建第二个字段集，其中将包含地址信息。以下代码清单代码量很大，所以请耐心看完。

代码清单6-8 构建第二个字段集

```
var fieldset2 = Ext.apply({}, {                ❶ 从第一个字段集复制属性
    flex       : 1,
    labelWidth : 30,
    title      : 'Address Information',
    defaults   : {
        layout : 'column',
        anchor : '100%'
    },
    items      : [
        {
            fieldLabel : 'Address',
            name       : 'address'
        },
        {
```

```
                fieldLabel : 'Street',
                name       : 'street'
            },
            {
                xtype : 'container',                    ❷ 添加列布局容器
                items : [
                    {
                        xtype       : 'fieldcontainer',       ❸ 添加输入框容器
                        columnWidth : .5,
                        items       : [
                            {
                                xtype      : 'textfield',     ❹ 配置州单行文本框
                                fieldLabel : 'State',
                                name       : 'state',
                                labelWidth : 100,
                                width      : 150
                            }
                        ]
                    },
                    {
                        xtype       : 'fieldcontainer',       ❺ 添加输入框容器
                        columnWidth : .5,
                        items       : [
                            {
                                xtype      : 'textfield',     ❻ 设置邮政编码单行文本框
                                fieldLabel : 'Zip',
                                name       : 'zip',
                                labelWidth : 30,
                                width      : 162
                            }
                        ]
                    }
                ]
            }
        ]
    }, fieldset1);
```

在代码清单6-8中，你使用Ext.apply❶方法，把fieldset1中的很多属性，复制到fieldset2中。这个工具方法被普遍用来复制或者覆盖一个对象或者另一个对象的属性。我们将在探讨Ext JS工具带的时候更多地讨论这个方法。要想得到期望的布局，并让州文本输入框和邮政编码文本输入框并排，你就必须借助于大量嵌套。第二个字段集的子项❷是一个容器，它的layout属性被设为'column'。该容器的第一个子元素是一个fieldcontainer❸，其中包含了州单行文本框❹。ColumnLayout容器的第二个子项❺是另一个fieldcontainer，其中包含了邮政编码单行文本框❻。

你可能在疑惑为什么有这么多嵌套的容器，以及为什么实现的代码会这么长。要想在其他布局内使用不同的布局，容器嵌套是必要的。这个道理你可能不会马上想明白，相信你在渲染表单的时候会更加明了。现在，还是让我们继续来为这两个字段集构建一个存在场所。

为了让表单能够并排显示，你需要为它创建一个容器，并将其设为使用HBox布局。为了让

HBox布局中的宽度相等，你必须把每个字段集的`stretch`属性都设为`1`。让我们来为两个字段集打造一个“家”：

```
var fieldsetContainer = {
    xtype        : 'container',
    layout       : 'hbox',
    layoutConfig : {
        align    : 'stretch'
    },
    Items        : [
        fieldset1,
        fieldset2
    ]
};
```

这段代码创建了一个容器，它的高度是固定的，但却没有设置宽度。这是因为这个容器的宽度将会通过VBox布局自动设置，它是你未来创建的表单面板所用的布局。

6.7.3　创建标签面板

接下来要构建一个标签面板，带三个选项卡，一个包含电话号码表单元素，另外两个包含HTML编辑器。这个标签面板将使用表单面板可用高度的下半部分。你将一次性地配置所有选项卡，所以下面这个代码清单很长。

代码清单6-9　构建一个带表单元素的标签面板

```
var tabs = [
    {
       xtype     : 'fieldcontainer',       <- ❶ 为单行文本框添加容器
       title     : 'Phone Numbers',
       layout    : 'form',
       bodyStyle : 'padding:6px 6px 0',
       defaults  : {
          xtype : 'textfield',
          width : 230
       },
       items: [
          {
            fieldLabel : 'Home',
            name       : 'home'
          },
          {
            fieldLabel : 'Business',
            name       : 'business'
          },
          {
            fieldLabel : 'Mobile',
            name       : 'mobile'
          },
          {
            fieldLabel : 'Fax',
```

```
            name       : 'fax'
          }
        ]
      },
      {
        title : 'Resume',
        xtype : 'htmleditor',
        name  : 'resume'
      },
      {
        title : 'Bio',
        xtype : 'htmleditor',
        name  : 'bio'
      }
];
```

❷ 添加两个HTML编辑器作为选项卡

代码清单6-9构建了一个数组，由三个选项卡组成，它们将是未来标签面板的子元素。第一个选项卡❶是一个有4个单行文本框的`fieldcontainer`。第二个❷和第三个选项卡是HTML编辑器，它们将被用来输入一篇摘要和一篇短传记。让我们继续来构建标签面板：

```
var tabPanel = {
    xtype              : 'tabpanel',
    activeTab          : 0,
    deferredRender     : false,
    layoutOnTabChange  : true,
    border             : false,
    flex               : 1,
    plain              : true,
    items              : tabs
}
```


下个代码清单中的任务是构建表单面板本身,而相比较构建它所有的子项,这一步要相对琐碎：

代码清单6-10 整合

```
var myFormPanel  = Ext.create('Ext.form.Panel', {
    renderTo     : Ext.getBody(),
    width        : 700,
    title        : 'Our complex form',
    frame        : true,
    id           : 'myFormPanel',
    layout       : 'vbox',
    layoutConfig : {
        align : 'stretch'
    },
    items        : [
        fieldsetContainer,
        tabPanel
    ]
});
```

终于可以创建表单面板了。设置`renderTo`以确保表单面板能被自动渲染。为了妥善设置`fieldsetContainer`和标签面板的尺寸，要使用VBox布局，并把`layConfig`的`align`属性设为

'stretch'。通过图6-17观察这个非常复杂的表单是如何渲染的。

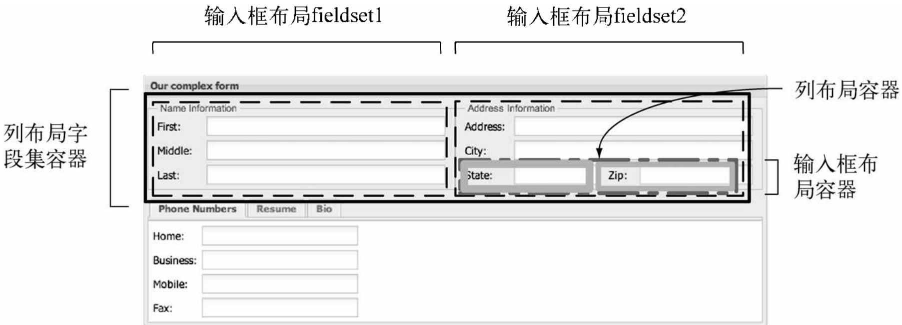


图6-17　你的第一个复杂布局表单的渲染结果，表单借助不同的容器组合出了复杂的布局

在图6-17中，我们把构成表单前半部分的不同容器设为高亮，其中包括fieldset Container，两个字段集和它们的子组件。通过使用这么多容器，你可以确保全盘控制UI的布局方式。用这些长段的代码来创建一个如此复杂程度的UI是很常见的。在试用新创建的表单面板的时候，你可以在三个选项卡之间切换，点出隐藏在下面的HTML编辑器。

到现在，你已经看到如何通过组合多个组件和布局得到既好用又节省空间的UI。接下来，让我们集中学习如何使用表单提交和加载数据，否则表单将毫无用处。

6.8　数据提交和加载

通过基本表单提交方法来提交数据是新开发者最经常碰壁的领域之一。这是因为很多年以来，我们已经习惯了在提交表单后期待页面刷新。而使用Ext JS，表单提交需要一些技巧。同样，用数据加载一个表单对有些人来说可能有点弄不懂，所以我们要探索几种可能的实现方式。

6.8.1　提交表单的传统方式

提交表单的传统方法是非常简单，不过需要配置表单面板的底层表单元素，把standard-Submit属性设为true。为了执行提交动作，调用：

```
Ext.getCmp('myFormPanel').getForm().submit();
```

这段代码会调用一般的DOM表单submit方法，它会用传统方式提交表单。如果要在表单面板中用这种方法，我们建议通过Ajax检查表单提交，这可以让你看到一些在使用传统表单提交方法时无法使用的功能。

6.8.2　通过Ajax提交数据

要提交一个表单，必须访问表单面板的BasicForm组件，为此使用访问方法getForm或者formPanel.getForm()。这样，就可以访问到BasicForm的submit方法，你将用它来通过Ajax

发送数据。代码如以下代码清单所示。

代码清单6-11 提交表单

```
var onSuccessOrFail = function(form, action) {
    var formPanel = Ext.getCmp('myFormPanel');
    formPanel.el.unmask();
    var result = action.result;
    if (result.success) {                                          ❶ 显示JSON驱动的信息
        Ext.MessageBox.alert('Success',action.result.msg);
    }
    else {
        Ext.MessageBox.alert('Failure',action.result.msg);
    }
};

var submitHandler = function() {
    var formPanel = Ext.getCmp('myFormPanel');
    formPanel.el.mask('Please wait', 'x-mask-loading');
    formPanel.getForm().submit({                                   ❷ 执行表单提交
        url     : 'success.true.txt',
        success : onSuccessOrFail,
        failure : onSuccessOrFail
    });
};
```


在代码清单6-11里，创建了一个叫onSuccessOrFail的成功和失败处理程序，如果表单提交的尝试成功或失败就将调用该处理程序。它会根据从网页服务器返回的JSON状态显示一个警告消息框❶。然后创建提交处理程序方法submitHandler，它执行表单提交动作❷。你可以在BasicForm或者表单面板这一层级指定URL，但之所以这里在调用提交的时候指定，是因为我们希望特别指出，目标URL可以在运行时间更改。另外，如果你想像这里一样，提供任何类型的等待消息，就应该有成功和失败处理程序。

最起码，返回的JSON应该包含一个值为true的'success'布尔值变量。成功处理程序预计还有一个msg属性，其中应该包含一个字符串，把一条消息返回给用户：

```
{success: true, msg : 'Thank you for your submission.'}
```

同样，如果服务器端代码认为这次提交出于任何原因没有成功，服务器应该返回一个success属性设为false的JSON对象。

如果想执行服务器端校验（可能会报错），那么返回的JSON也可能包括一个errors对象。下面是一个附有errors对象的失败信息示例：

```
{
    success : false,
    msg     : 'This is an example error message',
    errors  : {
        firstName : 'Cannot contain "!" characters.',
        lastName  : 'Must not be blank.'
    }
}
```

如果返回的JSON包含一个`eorrs`对象，那么通过该名称识别的所有输入框都会被标示为无效。图6-18展现了服务器端提供JSON代码的表单。

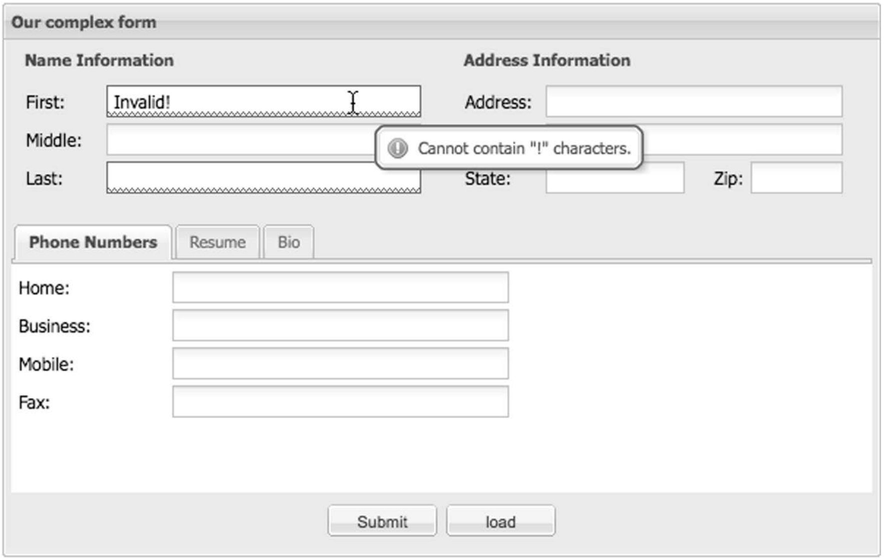


图6-18　服务器端`errors`对象使用标准的QuickTip报错`msg`后的运行结果

本节中，你知道了如何使用标准提交方法以及Ajax方法提交表单，还看到了如何使用`errors`对象，借助于UI层级的报错提示，提供服务器端校验。接下来，我们要学习如何使用`load`和`setValues`方法把数据加载到表单中。

6.8.3　把数据加载到表单中

几乎所有表单的使用周期都包含保存和加载数据。在Ext JS中，有几种方法可以加载数据，但必须有数据可以加载，所以首先要创建一些数据。让我们创建一些虚拟数据，并将其保存在一个名为data.json的文件里：

```
{
    "success" : true,
    "data" : {
        "firstName" : "Jack",
        "lastName"  : "Slocum",
        "middle"    : "",
        "address"   : "1 Ext JS Corporate Way",
        "city"      : "Orlando",
        "state"     : "Florida",
        "zip"       : "32801",
        "home"      : "123 345 8832",
        "business"  : "832 932 3828",
        "mobile"    : "123 332 2122",
```

```
        "fax"       : "392 322 9321",
        "resume"    : "Skills:<br><ul><li>Java Developer</li><li>Ext JS
Senior Core developer</li></ul>",
        "bio"       : " Jack is a stand-up kind of guy.<br>"
    }
}
```

和使用表单提交一样，根JSON对象必须包含一个值为`true`的`success`属性，它将触发`setValues`方法调用。另外，用于加载到表单上的值必须要在一个引用属性为`data`的对象里。同样，让你的表单元素名称和要加载的数据属性相对应是一个好方法。这样可以确保相应的输入框中填入正确的数据。要让表单通过Ajax加载数据，可以调用`BasicForm`的`load`方法，其语法跟`submit`方法类似：

```
var formPanel = Ext.getCmp('myFormPanel');
formPanel.el.mask('Please wait', 'x-mask-loading');
formPanel.getForm().load({
    url     : 'data.json',
    success : function() {
        formPanel.el.unmask();
    }
});
```


运行这段代码将导致表单面板执行一次Ajax请求来取得数据，而最终表单中将填入相应的值，如图6-19所示。

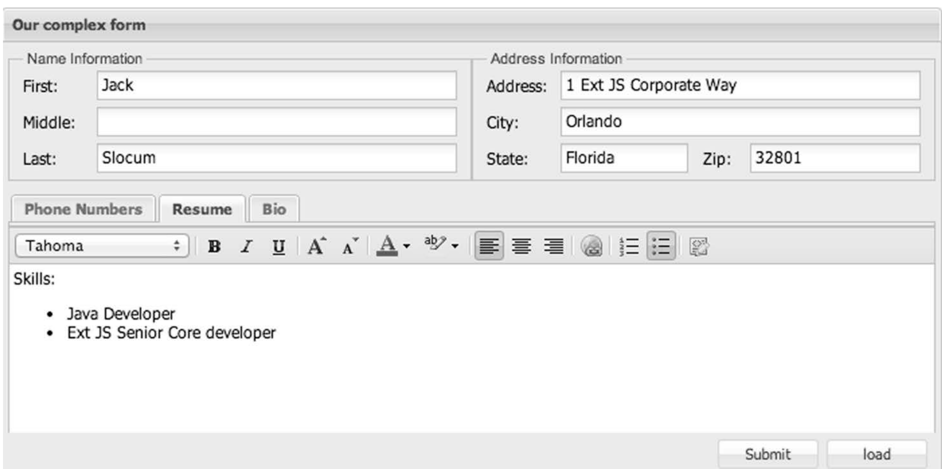


图6-19　通过Ajax加载数据的结果

如果你手头有数据，例如从另一个诸如数据网格的组件那里得到的，那么可以通过`myFormPanel.getForm().setValues(dataObj)`来设置值。使用这个方法，`dataObj`将会只包含对元素名称的正确映射结果。同样，如果你有一个`Ext.data.Record`的实例，则可以用表单的`loadRecord`方法来设置表单的各项值。

提示　要从任何特定的表单取得值，就调用`FormPanel`实例的`getValues`方法。举例来说，`myFormPanel.getValues()`会返回一个对象，里面包含的键代表着输入框的名称和它们的值。

加载数据可以如此简单。请记住，如果服务器端想拒绝数据加载，那么可以把`success`值设为`false`，这会触发在加载的配置对象中引用的`failure`方法。

祝贺你！你已经配置了第一个真正复杂的表单面板，并学会了加载和保存它的数据。

6.9　小结

对`FormPanel`进行的集中讨论涉及了很多主题，包括很多常用的字段。你甚至有机会深入了解了组合框，并在此过程中首次接触到了它的辅助类`dataStore`和`DataView`。借此，你知道了如何自定义组合框生成的列表框。此外，你还构建了一个相对复杂的布局表单，并使用新工具提交和加载了数据。

在此之后，我们将深入探讨数据网格面板。你将了解它的内部组件，知道如何自定义一个网格的观感。你还将学会使用网格面板的编辑器插件，以便直接编辑数据。在此过程中，你还将了解更多关于数据存储的知识。请准备好享受这次有趣的旅行吧。

第 7 章

数据存储

7

本章内容

- 使用数据存储
- 了解数据代理
- 探究写入器和校验

要创建一个实际应用，需要有一种持久存储数据的方法。数据持久化让应用的用户可以在不同会话之间访问数据，所以数据应该储存在一种媒介中，在应用被停止和重新启动后仍以被访问。

如果编写的是简单应用，就可以利用Ajax技术通过屏幕上的组件更新和检索数据，以此实现数据持久化。但如果创建的是涉及客户端的高阶交互逻辑的Ext JS应用，那么可以使用Ext JS数据存储中的可用功能。

本章首先带我们大体了解数据包。你将了解`Ext.data.Store`和它的支持类，包括`Ext.data.Model`，以及数据是如何流动又是如何被数据存储使用的。此外，我们将讨论不同的数据读取器，并通过数组、JSON和XML数据来探索数据使用。

你将熟悉所有的数据代理，学会从常驻内存、Ajax、JSONP和LocalStorage中读取数据。在本章的结尾，我们将通过演示如何处理数据校验和关联来介绍Ext JS 4数据包的高阶特性。

通读本章，你将拥有充足的知识与自信使用Ext JS框架创建任意数据驱动的视图，首先就是第8章的网格面板。

7.1 介绍数据存储

`Ext.date.Store`的用途是提供一个基础，可以存储服务器上数据的一个本地子集，并在将数据发回服务器之前跟踪这一数据的变化（如果应用允许编辑数据的话）。

数据存储在框架中任何需要的地方为很多部件供给数据。为了更直观地加以描述，图7-1列举了依赖`dataStore`类的类。

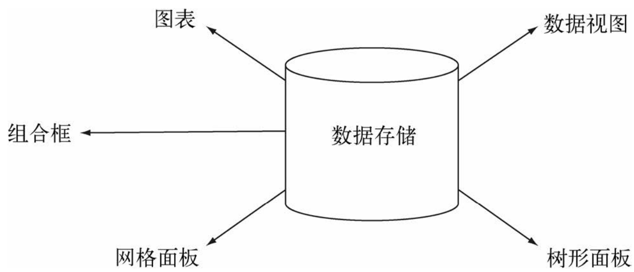


图7-1　数据存储和它供给数据的类。这张图并没有展示类继承关系

可以看到，数据存储支持很多种部件，包括数据视图、组合框、图表、网格面板和树形面板。数据存储还是树形面板的基础（我们将在第10章介绍树存储）。

`data Store`类在Ext JS框架和应用中扮演着不可或缺的角色，所以当然很有必要来详细介绍这个类。正因为此我们要用接下来的这几页来讨论`data Store`类，以及使用运用，然后再去尝试一些代码。

7.1.1　支持类

经常用数据存储来读取数据，这也很合理：`data Store`类类似于一个接口类。数据存储本身负责组织列举数据，但它主要是管理其他类。这些其他类支持数据存储，帮助它为视图提供数据以渲染在屏幕上。

为了理解支持数据存储的类，让我们先退一步，来看看一张数据包的简化视图。图7-2显示出`data Store`类与很多类有联系。每个类在框架对数据的使用中起到相应的作用。表7-1给出了这些类的用途。

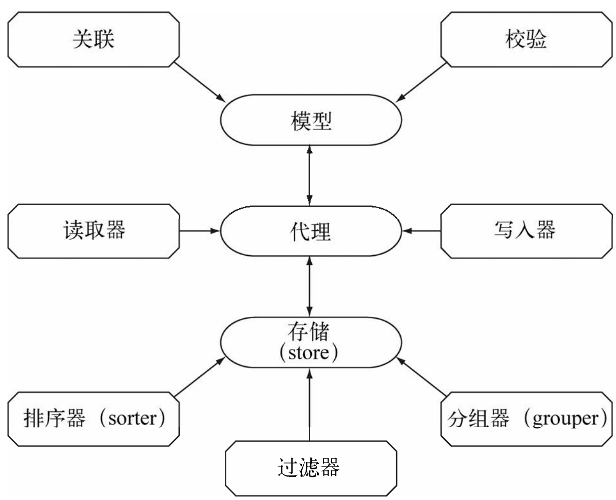


图7-2　数据包一览

表7-1 Data Store及其支持类

类	用　途
Store	通常用于读取和保存数据的接口类
Sorter	负责给数据排序
Filter	管理数据过滤
Grouper	用来对数据分组
Proxy	管理数据被输入数据存储的典型方式
Reader	用来转化入站的数据，以便供给Model的实例
Writer	负责组织列举数据以便给发往数据存储进行存储
Model	代表对应一个特定数据存储的独立数据行
Association	允许模型通过预配置的规则彼此之间关联
Validation	防止模型被不正确或者不完整的数据破坏的一种方法

框架为数据存储提供了大量支持。这些类的设计目的是为了让你的工作更轻松，帮助应用在浏览器中方便地管理健壮的数据。

对于数据存储是如何使用支持类的，你现在已经略知一二。知道数据是如何流动的，这在开始使用数据存储的时候将会很有帮助。

7.1.2 数据是如何流动的

让我们来看看数据是如何从一个来源流动到数据存储的。我们首先来看一个基本的流动，如图7-3所示。

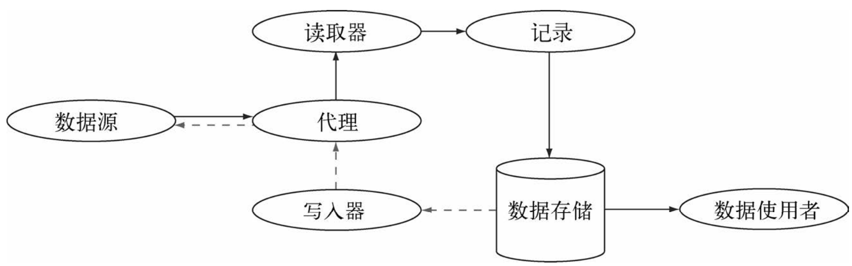


图7-3 从一个数据源到数据存储使用者的数据流动

可以看到，数据总是源自一个数据源，而数据的读取和保存是由一个数据代理管理的。数据Proxy类简化了从众多来源对未格式化数据对象的获取，并借助于诸如数据Reader这样的定制类包含它们自己用于彼此通信的事件模型。

比如说，在一个与数据使用者（比如网格面板）绑定的数据存储中改变一个模型的值，将导致UI在该模型被执行的时候更新。在模型被加载到数据存储后，绑定的数据使用者会刷新它的试图，读取周期完成。

特定的data Store子类将会拥有load、remove和sync操作。load和remove操作处理数

据存储里的本地模型列表，而`sync`操作通过选定的代理把数据存储的本地内容和数据端点同步。

刚刚已经看到，数据代理对于数据存储的操作至关重要。让我们深入了解什么是数据代理，以及有哪些种类的代理可供使用。

7.1.3　关于数据代理

在框架中有一个命名很恰当的抽象类`Ext.data.proxy.Proxy`，它是多个子类的基类，负责从特定的来源获取数据并对其写入数据，如图7-4所示。

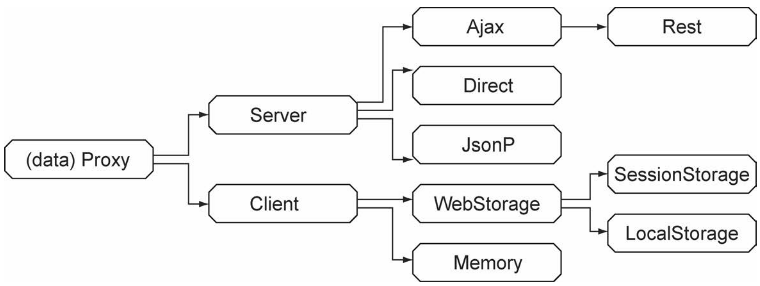


图7-4　数据代理和它的7种特定实现方式，每一种都负责从一个特定的数据源获取数据

这么多不同的代理刚开始可能让你有点儿眼花缭乱。如果你是从Ext JS 3过渡来的，那这看起来简直像是数据包大爆炸！可当你发现Server、Client和WebStorage代理是基类，只有剩下的7个类可以实际使用的时候，就可以放松下来了。在这里我们将逐个地详细探讨这7个类、

最常用的代理是Ajax代理，它使用浏览器的XHR对象来执行一般Ajax请求。Ajax代理被限制在同一个域内，因为所谓的同源策略。该策略决定了通过XHR提交的XHR请求不能在一个特定页面加载的域以外执行。这一策略的目的是为了加强使用XHR的安全性，但被普遍认为与其说是个安全措施，还不如说是个麻烦东西。Ext JS开发人员很快就为这项“特性”开发了一个解决办法，这时JsonP代理登场了。

JsonP代理聪明地使用脚本标签从另一个域获取数据，它的效果很好，但需要提出请求的域返回JavaScript代码而不是一般的数据片段。了解这一点很重要，因为不能对任何第三方网站使用JsonP代理来获取数据。Json代理需要把返回的数据封装在一个全局方法调用中，把数据作为唯一参数传入。稍后你将进一步学习JsonP代理，因为将用它配合extjsinaction.com上多种不同的API从我们的示例中获取数据。

`MemoryProxy`类让Ext JS可以从一个内存对象中读取数据。虽然可以通过一个`data Store`实例的`loadData`方法把数据直接读取到该实例，但使用`MemoryProxy`类在某些情况下很有帮助。示例之一就是重新读取数据存储的任务。如果使用`Store.loadData`，就需要传入该数据的引用，后者由读取器解析，并被读入数据存储。使用内存代理可以让事情更简单，因为只需要调用`Store.reload`方法，并让Ext JS来处理苦差事。

Direct代理使得数据存储可以与`Ext.direct`远程调用提供者交互，从而允许通过远程方法

调用（RPC）来获取数据。

注意　如果你有兴趣更深入地了解Ext.Direct，大可以先看一下第11章，不过看过后马上回来。我们将在这里介绍很多你理解第11章需要的基础知识。

RestProxy是AjaxProxy的子类，专门用于跟Rest式资源交谈。比如说，如果想获取ID为403的员工资料，Rest代理会自动发送一个GET请求到/employee/403。这一自动URL创建在一个已经使用这一系统的环境中会很方便。

当应用离线或者当你不希望与服务器交谈的时候，可以选择使用LocalStorage代理或者SessionStorage代理来存储数据。这些客户端代理通过HTML5 Web存储API采用现有的“键/值”机制来持续保存数据，所以相关的数据结构将被自动序列化为JSON。如果浏览器上不具备HTML5网页存储API，那代理会弹出一个异常。如果想在会话之间持续保存数据，就使用LocalStorage代理；如果只想在浏览器会话激活的时候保存数据，那就使用SessionStorage。

注意　请记住，需要给每个WebStoage代理提供唯一的ID。

7

所有代理都实现了相同的接口，理论上应该是可以互换的。不过有几点要注意。因为JavaScript的限制，在用JsonP代理进行写入时所有参数都是用GET方法传入。这并非不合理，但如果选择换用JsonP代理，那么在不得不重新实现初始使用Ajax代理的后台程序时，这可能会带来“惊喜”。另外也应该很显然的一点是，在从Ajax代理转用LocalStorage代理时，没有理由保留一个url设置。

7.1.4　模型和读取器

数据存储的基础是Ext.data.Model。它把数据保留在一个字段列表中，并用Ext.data.Association描述与其他模型的关联，而且它允许使用Ext.data.validations来校验。可以使用Ext.data.Reader把数据读入数据集，并用Ext.data.Writer把数据写回到服务器。

如果你熟悉SQL数据库，那可能会注意到Ext.data.Model跟一个SQL数据库中的表格有些类似。每次实例化一个特定的Ext JS数据模型，可以说都是在模拟客户端SQL数据库中一个条目的内容（请记住，不要求你在客户端保存整个数据集）。

虽然使用数据存储的典型情况涉及服务器端的SQL数据库，但如果浏览器中有LocalStorage或者SessionStorage，那数据存储也能使用它们保存数据。把Ext JS数据存储想象成一个API，把当前的数据传输选择（JSON、XML或者LocalStorage）想象成API的一个实现。

在深入窥探数据存储之前，让我们首先来定义一下你拥有的数据使用选项。在一个代理获取到原始数据后，一个读取器对数据进行读取或者解析。读取器从未格式化的原始数据对象中提炼

出数据点，也就是所谓的数据索引，并把它们组织成名称数据对，或者一般对象。图7-5展现了这一映射如何使用。

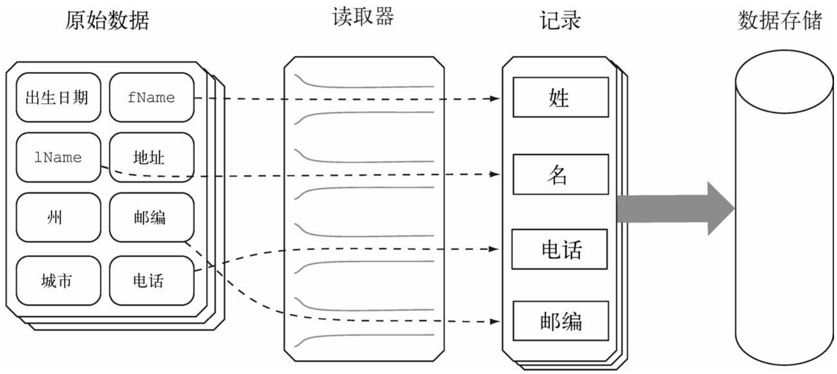


图7-5　一个读取器映射原始数据或者未格式化数据，使之可以被插入模型，后者然后被导入一个数据存储

可以看到，未格式化的原始数据被组织并传入读取器创建的`Model`实例。这些`Model`类的实例然后被导入数据存储中，准备被一个部件使用。

注意　Ext JS 3的`Ext.data.Record`也就是Ext JS 4中的`Ext.data.Model`。这个名称改变更直接地确立了本书第三部分描述的“模型－视图－控制器”模式中数据存储的模型部分。

一个`Model`是一个完全由Ext JS管理的JavaScript对象。跟Ext JS管理`Element`类很相似的是，`Model`类也有获取方法和设置方法，以及数据存储绑定的一个完整的事件模型。这种数据管理方式可以给框架增加可用性和一些很酷的自动功能。

7.4节将介绍如何使用获取方法和设置方法通过数据存储更新模型数据，但首先我们要为数据的更新打好基础，描述数据写入器在其中如何发挥作用，以及如何在与数据存储交互以前校验你的数据。

`Reader`这个类负责协助转化从数据源输入的数据。它转化输入的数据，并实例化`Model`定义的实例，后者由数据存储收集。读取器的类型有很多，而在本章中你要把它们逐个实现。

7.2　读取和保存数据

有多种不同的使用数据的选项。每种选项都有着不同程度的复杂性，所以在开始使用数据存储以前应该仔细考虑需求。所选择的数据传输机制会影响需要用到`Ext.data.Proxy`、`Ext.data.Reader`和`Ext.data.Writer`的哪些子类。

在读取数据时，请记住数据端点是把`Ext.data.Field`用作一个数据存储抽象的`Ext.data.model`。所以当读取完数据时，读取数据所采取的步骤变得与其他的应用代码不相干。这就意

味着在后来的阶段中，如果应用一开始使用XML来作传输介质的话，那么可以选择用JSON来代替XML。

7.2.1 读取数组数据

数据存储最简单的使用方式是把内联数据读入到一个组件里。试想一种典型的使用情况，给一个人选择一个头衔。可以使用一个组合框方法来实施，把头衔列表置入像这样的一个内联数据数组：

```
{
    Xtype      : 'combo',
    name       : 'title',
    fieldLabel : 'Title',
    queryMode  : 'local',
    valueField : 'title',
    store      : ['Mr.', 'Ms.', 'Dr.']
}
```

请注意store声明在这里只占了单独的一行，而其他行代码与组合框有关。如果这是你第一次接触Ext JS，那这个示例可能会令人有点儿困惑。Ext JS提供了很多快捷方式和便利类，而这是一个最佳的示例。

组合框示例没有一个model或者fields定义。在数据存储的构建阶段，Ext JS会创建Ext.data.ArrayStore，后者会自动替你调用Ext.data.reader.Array，并设置它以便用于运行时间。

让我们更深入地探讨数据数组存储和数组读取器，以更熟悉数组数据的使用过程，如以下代码清单所示。

代码清单7-1　创建一个读取本地数组数据的数据存储

```
var arrayData = [                          ❶ 创建本地数组数据
    ['Jay Garcia',     'MD'],
    ['Aaron Baker',    'VA'],
    ['Susan Smith',    'DC'],
    ['Mary Stein',     'DE'],
    ['Bryan Shanley',  'NJ'],
    ['Nyri Selgado',   'CA']
];

Ext.define('User', {                       ❷ 创建用户模型
    extend : 'Ext.data.Model',
    fields : [                             ❸ 声明字段
        {
            name    : 'name',
            mapping : 1
        },
        {
            name    : 'state',
            mapping : 2
```

```
            }
        ]
    });

    store = Ext.create('Ext.data.Store', {           ◁ ❹ 创建数据存储
        model : 'User',
        proxy : {
            type   : 'memory',
            reader : {
                model : 'User',
                type  : 'array'
            }
        }
    });

    store.loadData(arrayData);                       ◁ ❺ 把数据读入数据存储
    console.log(store.first().data)
```

在代码清单7-1中，实现了全部的数据存储配置。首先创建一个数组的数组，后者通过变量arrayData加以引用❶。请密切关注数组数据所采用的格式，因为这是ArrayReader类的预期格式。数据之所以是一个数组的数组，原因在于包含在父数组里的每个子数组都被视为一条单独的记录来对待。

创建一个用户模型❷，这个模型将被用作模板来映射数组数据点以创建记录。把一个对象常量的数组，也就是fields❸，传入配置对象，并详细列出每个字段名及其映射。这些对象常量每一个都是Ext.data.Field类的配置对象，后者是Ext.data.Model中最小的被管理数据单元。在这里，把字段名映射到每条数组记录的第一个数据点上，并把字段状态映射到第二个数据点上。

接下来，用一个Memory代理创建一个data Store的实例，这个Memory代理将会从内存中读取未格式化的数据❹。

然后添加一个ArrayReader实例❺，它负责整理该代理获取到的数据，并且创建刚刚创建的新用户模型的新实例。在数据存储读取数据的时候，数组读取器读取每条记录，并创建一个新的User实例，把解析过的数据传给该实例，然后把它读入数据存储。

以上就是端对端创建一个数据存储读取数组数据的示例。利用这一模式，可以改变该数据存储可以读取的数据的类型。为此，要用一个JsonReader或者XmlReader来代替ArrayReader。同样，如果想改变数据存储，可以把MemoryProxy换成其他代理，比如Http Proxy、JsonP Proxy或者Direct Proxy。

回想一下，之前提到过便利类可以让我们的工作更轻松。如果要用ArrayStore便利类重建数据存储，使用之前的arrayData的代码将会是这样：

```
var store = Ext.create('Ext.data.ArrayStore', {
  data   : arrayData,
  fields : ['personName', 'state']
});
```

可以看到，在这个示例中用字段的快捷符号来创建一个Ext.data.ArrayStore的实例。为此，要传入一个数据的引用（也就是arrayData），并传入一个字段列表，由它来提供映射。注意，fields属性是一个简单的字母串列表。这是一个完全有效的字段映射配置，因为Ext JS足够智能，可以根据以这种方式传入的字符串值来创建名称和索引映射。还可以在一个fields配置数组中混合对象和字符串。比如说，以下配置完全有效：

```
fields : [
    'fullName',
    {
        name    : 'state',
        mapping : 2
    }
]
```

像这样的灵活使用会很酷。不过要知道使用这样的混合字段配置可能会略微有损代码的易读性。

使用这个便利类可以不用非得创建代理，记录模板和读取器来配置数据存储。使用JSON数据存储或者XML数据存储也同样简单，稍后你就将了解到。接下来，将使用便利类来节省时间。

7.2.2 读取JSON 数据

JSON存储要比数组存储更复杂一点，因为它也可以读取相关的数据结构，并从服务器获取数据。很多人选择用服务器端栈提供JSON数据，因为这样浏览器更容易消化数据。

我们假设从服务器端获取到了这样一个JSON格式的部门列表。

```
{
    "data" : [
        {
            "id"            : "1",
            "name"          : "Accounting",
            "active"        : null,
            "dateActive"    : "12/01/2001",
            "dateInactive"  : null,
            "description"   : null,
            "director"      : null,
            "numEmployees"  : "45"
        }
    ],
    "meta" : {
        "success" : true,
        "msg"     : ""
    }
}
```

当跟服务器对话时，获取数据的过程中可能会出现某些问题（比如，数据库可能出故障）。所以要在响应后发送信息以确认该响应是否成功，并在出错的时候发送一条自定义出错信息。这个方法可以轻易地更换出错信息，而不用担心服务器上的特定设置细节（如果在proxy上设置

successProperty属性，那出错处理是自动完成的，如代码清单7-2所示）。

以下代码清单显示了如何使用JsonStore便利类来读取JSON。可以在examples/ch07里找到这个代码清单里指定的data.json文件。

代码清单7-2　读取JSON数据

```
var departmentStore = Ext.create('Ext.data.Store', {      <-- ❶ 实例化JsonStore
    fields : [
        'name',
        'active',
        'dateActive',
        'dateInactive',
        'description',
        'director',
        'numEmployees',
        {
            name : 'id',
            type : 'int'
        }
    ],
    proxy : {                                              <-- ❷ 选择Ajax代理
        type : 'ajax',
         url    : 'data.json',                             <-- ❸ 设置URL
        reader : {                                         <-- ❹ 配置JsonReader
            type            : 'json',
            root            : 'data',
            idProperty      : 'id',
            successProperty : 'meta.success'
        }
    }
});

departmentStore.load({
    callback : function(records, operation, successful) {
        if (successful) {
            console.log('department name:',                <-- ❺ 打印第一条记录
                                   records[0].get('name'));
        }
        else {
            console.log('the server reported an error');
        }
    }
});
```

首先设置一个带field定义的JsonStore❶，然后选择一个Ajax代理❷，设置用于联系的服务器url❸，并且配置JsonReader（注意，这个示例所指的是本地安装地址；请在本书所附的示例中查找一个示例列表，地址http://extjsinaction.com/v4/examples/ch07）。把读取器类型设为json，把root❹设为data。配置reader的时候，设置idProperty和successProperty属性，以跟踪服务器更新时的变化，稍后即可看到。Ext JS需要idProperty，以便在发送更新回服务器以前进行内部簿记。最后，打印出服务器获取的列表的第一条记录❺。

7.2.3　读取XML数据

还可以选择把XML数据读入数据存储。我们假设部门列表是XML格式的：

```
<?xml version="1.0" encoding="UTF-8" ?>
<Response>
  <data>
    <node>
      <id>1</id>
      <name>Accounting</name>
      <active>true</active>
      <dateActive>12/01/2001</dateActive>
      <dateInactive></dateInactive>
      <description>Accounting services</description>
      <director></director>
      <numEmployees>45</numEmployees>
    </node>
    ...
  </data>
  <meta>
    <success>true</success>
    <msg></msg>
  </meta>
</Response>
```

接下来这个代码清单展示了如何把XML格式的数据读取转化为在之前示例中使用的字段。可以在examples/ch07中找到data.xml文件。

代码清单7-3　读取XML数据

```
var departmentStore = Ext.create('Ext.data.Store', {      ❶ 使用XmlStore
  fields : [                                               ❷ 声明字段
      'name',
      'active',
      'dateActive',
      'dateInactive',
      'description',
      'director',
      'numEmployees',
      {
          name    : 'id',
          mapping : 'id'
      }
  ],
  proxy : {
      type   : 'ajax',
      url    : 'data.xml',
      reader : {
          type            : 'xml',                         ❸ 选择XmlReader
          record          : 'node',
          idPath          : 'id',
          successProperty : 'meta/success'
      }
```

```
        }
    });
    departmentStore.load({
        callback : function(records, operation, successful) {
            console.log(operation)
            if (successful) {
                console.log("department:%o", records[0]);
            }
            else {
                console.log("the server reported an error");
            }
        }
    });
```

为读取XML，要选择一个`XMLStore`❶，并声明`fields`❷，类似于在之前示例中的做法。读取XML时，需要一个`XMLReader`❸，且必须使用`record`属性声明在哪里可以找到XML中的记录数据。正如在之前的示例中，设置`idPath`和`successProperty`可能会有用处。

你已经学习了如何把数据读入数据存储，现在是时候学习如何写数据了。

7.3　带写入器的数据存储

`Writer`可以节省时间和精力，让你无需再编写Ajax请求和异常处理，让你有更多时间做那些重要的事，比如为应用构建业务逻辑。在开始编程实现`Writer`之前，应该先评估`Writer`在其中如何适用。

要使用`Writer`，需要重新配置数据存储以及支持的代理。不是为该代理配置一个`url`属性，而是要创建一个配置对象`api`。代理`api`是一个新概念，在稍后我们回顾示例代码的时候将详细讨论。

需要创建一个`Writer`实例，并把它插入数据存储，同时给数据存储的配置对象本身添加一些新的配置属性，如以下代码清单所示。

代码清单7-4　Employee数据存储

```
var urlRoot = 'http://extjsinaction.com/crud.php?model=Employee&method=';
var employeeStore = Ext.create('Ext.data.Store', {
    model : 'Employee',                                   ❶ 引用Employee模型
    proxy : {
        type   : 'jsonp',                                 ❷ 创建新代理
        api    : {
            create  : urlRoot + 'CREATE',                 ❸ 配置代理API
            read    : urlRoot + 'READ',
            update  : urlRoot + 'UPDATE',
            destroy : urlRoot + 'DESTROY'
        },
        reader : {
            type            : 'json',                     ❹ 设置JSON读取器
            root            : 'data',
            idProperty      : 'id',
            successProperty : 'meta.success'
```

```
        },
        writer : {
            type            : 'json',          ⑤ 配置写入器
            encode          : true,
            writeAllFields  : true,
            root            : 'data',
            allowSingle     : true,
            batch           : false,
            writeRecords    : function(request, data) {
                request.jsonData = data;
                return request;
            }
        }
    }
});
employeeStore.load();
```

在代码清单7-4中，创建一个引用Exployee模型❶的数据存储，并使用一个带配置对象的Ajax代理❷作为属性api❸，后者为每一个CRUD操作指定了URL，而read就是读取数据的请求。你将使用有一定智能的远程服务器端代码，其中每个CRUD操作都有一个控制器。Writer需要有智能响应，因此，开发出了远程服务器端代码。使用相同的服务器端技术来处理所有CRUD操作是个好主意（请参考本书第三部分的示例应用，了解相关示例）。

注意 在这本书附送的示例集中，你会找到这个示例的一个版本，参阅了extjsinaction.com（employee_store.html）。这个示例会让你读取但并不更新数据。如果你希望了解执行CRUD操作的效果，那么建议遵照第7章附带的readme.txt中的指示。这个文件会教你在MySQL中设置数据库，以及配置附带的服务器端代码。

接下来创建一个Ext.data.Writer的子类，也就是Json Writer❹，它有可能可以保存一个请求来修改单条记录或者一批记录。这个示例展示了如何通过覆盖updateRecords❺，确保Writer一次写入一条记录。在JsonWriter配置对象中，把writerAllFields设为true，这可以确保每一次操作Writer都返回所有属性；这一策略很利于开发和调试。当然了，在开发中会希望把这个writerAllFields设为false，这会减少网络上和服务器端以及数据库栈的负担。

注意 可以在代理中覆盖Reader和Writer，Ext JS中大多数其他功能都可以覆盖。如果提供的类不符合你的需求，那很可能Ext JS开发社区中的某人已经实现了一个自定义类 。访问http://sencha.com/forum，看在开始实现你自己的自定义解决方案以前能找到什么。

我们建议用Writer进行开发时，给数据存储添加一个全局异常事件监听器。如果你希望对数据存储产生的任务异常都进行处理，那就需要这个监听器。在开发应用的时候，我们把所有参

数都发送到Firebug控制台，因为它提供了大量在调试过程中别处难以找到的信息。我们强烈建议你也这么做。相信我们，长远来看这么做会节省时间。

在为写入器实现服务器端数据端点的时候，你会注意到如果选择Json代理，那所有参数都是作为GET参数发送的。如果想在使用Ajax代理和JsonP代理之间转换的话，请牢记这一点。

7.3.1　校验模型数据

除了指定模型内字段的预期类型和格式以外，还可以指定在任何一次尝试更新数据存储之前运行的校验。在Ext JS 4中，可以直接用模型来校验字段，在此之前必须要在使用表单面板时执行校验。把校验代码放在靠近字段类型描述的地方，可以更清晰地指定域模型中的规则。

你可能还记得，`Ext.data.Field`可能包含：

- 一个名称；
- 一个类型（自动型、字符串型、整型、浮点型、布尔型、日期型）；
- 一个`defaultValue`；
- 一个转换函数（用来转换一条输入记录）。

当使用`Ext.data.Store`时，所有数据都传入`Ext.data.Model`，使得它成为一个执行数据校验的明显场所。在下面这个代码清单中，让我们通过设置一个代表某部门雇员的模型来探讨这一点。

代码清单7-5　带校验的雇员模型

```
Ext.define("Employee", {
    extend      : 'Ext.data.Model',
    fields      : [
        'firstName',
        'lastName',
        'middle',
        'title',
        'street',
        'city',
        'state',
        'zip',
        'departmentName',
        'rate',
        'officePhone',
        'homePhone',
        'mobilePhone',
        'email',
        {
            name : 'id',
            type : 'int'
        },
        {
            name : 'departmentId',           <-
            type : 'int'                      ❶ 转化为整型
        },
```

```
        {
            name   : 'dateHired',
            type   : 'date',
            format : 'Y-m-d'                          ❷ 指定dateHired
        },
        {
            name   : 'dateFired',
            type   : 'date',
            format : 'Y-m-d'
        },
        {
            name   : 'dob',
            type   : 'date',
            format : 'Y-m-d'
        }
    ],
    validations : [
        {
            type  : 'presence',                       ❸ 增加存在校验
            field : 'firstName'
        },
        {
            type  : 'presence',
            field : 'lastName'
        },
        {
            type  : 'presence',
            field : 'departmentId'
        },
        {                                             ❹ 配置格式校验
            type    : 'format',
            field   : 'email',
            matcher : /@/
        }
    ]
});
```

一种常见的使用情况是拥有一个id和几个外键，都为int型❶。还可以指定dateHired❷、dateFired和dob日期。

可以直接在模型上实施校验是Ext JS 4中的新功能。在这个示例中可以看到如何使用一个存在校验❸和一个格式校验❹。

假设要在第15号部门添加一个很年轻的雇员：

```
var sofie = Ext.create('Employee', {
    firstName     : 'Sofie',
    lastName      : 'Andresen',
    dob           : Ext.util.Format.date('2007/12/15','Y-m-d'),
    email         : 'Sofie A'
});
```

如果运行以下代码：

```
var errors = sofie.validate();
```

就会得到一张含有两条出错信息的列表。第一条出错信息显示Sofie还没有关联到一个部门，而之所以弹出第二条出错信息，是因为她才刚学会如何打她的名字，还不知道什么是电子邮件。所以让我们先不着急把Sofie添加到Employee数据存储。

如果用其他的数据成功地检验了模型数据而又没有出错，那就做好添加新雇员，与数据存储进行交互的准备了。

7.3.2　同步数据

你可能还记得图7-1中，数据存储理应是由使用者部件来使用的。因此对于真实的应用，要对部件的事件做出响应，并根据代码中的事件来操作。比如说接收到一个保存按钮的`onClick`事件，那就适宜使用部件中现有的数据来更新数据存储中相应的模型。

我们会在后面的几章中详细介绍一个使用者部件如何与数据存储交互。现在，让我们来探究如何用数据存储和模型中可用的指令来更新雇员数据。

注意　在学习如何与数据存储交互的时候，你可以直接在首选浏览器的开发工具中输入JavaScript示例代码片段。

首先可以用`set`和`get`来更新模型。数据包是有智能的，所以它知道在同步时什么时候会碰到需要更新的脏数据。

```
var firstEmployee = employeeStore.first();
firstEmployee.set('firstName', 'Anita');
firstEmployee.set('lastName', 'Andresen');
employeeStore.sync();
```

在这里获取Employee数据存储里第一个可用雇员模型的引用，把名改成Anita，把姓改成Andresen，并同步数据存储。这一步会激活代码清单7-4里提供的示例自定义写入器。

如果想给15号部门增加一个新雇员，可以创建以下雇员模型：

```
var jacob = Ext.create('Employee', {
    firstname    : 'Jacob',
    lastName     : 'Andresen',
    departmentId : 15,
    email        : 'jacob.andresen@gmail.com'
});
```

把它添加到Employee数据存储然后同步：

```
employeeStore.add(jacob);
employeeStore.sync();
```

如果对模型的一项校验操作失败，对新记录的同步操作就会失败。举例来说，如果你忘了给

Jacob记录分配一个departmentId。那这条记录就不会出现在调用sync方法创建的记录列表中。

> **注意** 可以在调用sync方法前调用数据存储的getNewRecords方法，查询要创建的是哪些记录。如果希望查询修改过的记录，可以调用getModifiedRecords方法，而getRemovedRecords方法显示同步时会移除哪些记录。

如果想移除Employee数据存储中的最后一条记录，那么可以这么做：

```
var lastEmployee = employeeStore.last();
employeeStore.remove(lastEmployee);
employeeStore.sync();
```

> **注意** 在Ext JS 4中，还可以直接使用模型的代理功能对模型执行读取、保存和销毁操作。请注意，确保在程序中保持编程风格的一致。如果混合使用两种编程风格，那代码可能会变得很难读。

如果你看过在把数据读入Employee数据存储时获取的数据，那么可能注意到了一个URL里的page、start和limit参数，看起来就像这样：

```
http://extjsinaction.com/
    crud.php?model=Employee&method=READ&_dc=1359395332718&page=1&start=0&lim
    it=25&callback=Ext.data.JsonP.callback1
```

page、start和limit参数都是系统提供的标准参数，可以被用来执行分页操作（把数据显示为较小的片段，而不是一次性显示全部数据）。我们将在第8章介绍分页组件的时候详细探究分页。

现在你对于什么是数据存储以及如何使用它应该有了实践知识。所以让我们用Ext JS 4中引入的一个高级特性来结束本章：那就是在模型之间使用Ext.data.Association嵌套数据的功能。

7.4 关联数据

通过嵌套数据，可以在运行高交互性应用的同时，满足少占用带宽的额外需求。它还可以以一种相比于使用Ext JS 3更准确的方式来表达业务逻辑。

试想人力资源部分配了任务，让你优化应用，使之能够支持快速修改公司所有部门的雇员数据。人力资源部主管渐渐受够了你的应用，他管它叫“传统型互联网应用”。“实在太慢了，”他说，“我浏览数据的时候它就只会跟服务器交互。我希望应用可以在我浏览所有部门的雇员列表的同时，即时地更新雇员数据。你能不能在应用启动时就读取所有数据？应用的启动时间长一点儿我无所谓，我只期望能用在我们的千兆内部网上。”

你可能很想告诉他，你这个分部门显示雇员列表的好主意是为了让应用渲染得更快，而且应用已经很快了，在抑制了如此争辩的冲动后，你开始寻找可能的选项。幸运的是，没过多久你就在文档里找到Ext.data.Model的描述，并发现它包含了Ext JS 4中的关联功能。看来要使用关联的数据读取，你必须要对现有代码进行一些微调，这样可以一次性地读入所有部门和所有雇员的描述。

首先增加一条从部门模型到雇员模型的hasMany关联，如以下代码清单所示。

代码清单7-6　带关联的部门模型

```
Ext.define('Department', {
    extend: 'Ext.data.Model',
    fields: [
        'id',
        'name',
        'active',
        'dateActive',
        'dateInactive',
        'description',
        'director',
        'numEmployees'
    ],
    sortInfo : {
         field : 'name',
         dir   : 'ASC'
    },
    associations: [{                     ❶ 包含hasMany 关联
        type  : 'hasMany',
        model : 'Employee',
        name  : 'employees'
    }]
});
```

在部门模型中，判断一个部门有很多雇员❶，把关联键设为employees。与此类似，要在代码清单7-5的employeeModel中设置一个belongsTo关联：

```
Ext.define("Employee", {
    extend: 'Ext.data.Model',
    fields: [ … ],
    associations: [{
        type           : 'belongsTo',
        model          : 'Department',
        associationKey : 'departmentId'
    }]
});
```

这么做，可以访问与当前部门相关联的雇员数据。下面这个代码清单使用代码清单7-5和代码清单7-6中的部门和雇员模型。

代码清单7-7 读取部门关联的雇员数据

```
var urlRoot = 'http://extjsinaction.com/crud.php?model=Department&method=';

var departmentStore = Ext.create('Ext.data.Store', {
    model : 'Department',
    proxy : {
        type : 'jsonp',
        api  : {
            create  : urlRoot + 'CREATE',
            read    : urlRoot + 'READ',
            update  : urlRoot + 'UPDATE',
            destroy : urlRoot + 'DESTROY'
        },
        reader : {
            type            : 'json',
            root            : 'data',
            idProperty      : 'id',
            successProperty : 'meta.success'
        }
    }
});

departmentStore.load({
    params : {
        detail : true,                          ❶ 要求细节
        limit  : 5
    },
    callback : function() {                          ❷ 读取相关数据
        departmentStore.each(function(department) {
            department.employees().each(function(employee) {
                var departmentId   = department.get('id'),
                    departmentName = department.get('name'),
                    employeeId     = employee.get('id'),
                    employeeName   = employee.get('firstName');

                console.log(departmentId, departmentName,
                            employeeId, employeeName);
            });
        });
    }
});
```

在这个示例中，要求示例的服务器端代码提供部门相关联的详细信息（与雇员模型相同的数据模型格式，只是针对部门进行了过滤）。参数`detail`❶被设为`true`，以表示服务器端代码应该提供对应部门的相关联雇员信息。

然后把`name`属性用作一个函数来使用相关联的雇员数据，在这个示例中也就是`department.employee()`❷。回想一下，代码清单7-6在部门模型的`hasMany`关联中提供了`name`属性。通过调用`department.employees()`，可获得了Employee数据存储的一个引用，用它来遍历与该部门相关的雇员。在这个示例中，我们选择了打印出部门名称和雇员的名字，这应该能证明可以访问相关联的雇员数据。

这就有了，使用Ext JS数据存储的嵌套数据。

7.5　小结

本章介绍了数据存储，以及很多支持数据存储的类。你看到了对各种不同支持类的概要分析，了解到代理负责获取数据，而读取器负责转化数据并创建`Model`实例以插入数据存储本身。

入门数据存储的世界之后，你深入学习了如何加载和使用Array、XML和JSON数据。然后，你知道了如何使用数据写入器和数据存储的`sync`方法持久保存数据。最后，你探究了数据包的数据关联功能，在此过程中读取了带相关雇员信息的部门数据。

在接下来的一章中，我们深入探究第一个复杂数据驱动视图：网格面板。

第 8 章

网格面板

8

本章内容

- 学习网格面板
- 使用已有的列实现
- 启用网格面板的分页功能
- 创建可编辑的网格面板
- 使用数据存储实现CRUD

从Ext JS早期开始，网格面板就是框架最核心的部分了。从很多方面来说，至今这一点依然成立。毋庸置疑，网格面板是Ext JS中功能最强大的部件之一,它通过表格形式提供了高速的数据访问。使用网格面板可以直观地查看并操作大数据集，这个类能够满足大部分企业和办公应用的需要。注意，在Ext JS 4.1中网格经历了一次重大的性能改进。现在网格非常快，以至于我们不再需要Ext JS之前发布版本中的列表视图了。

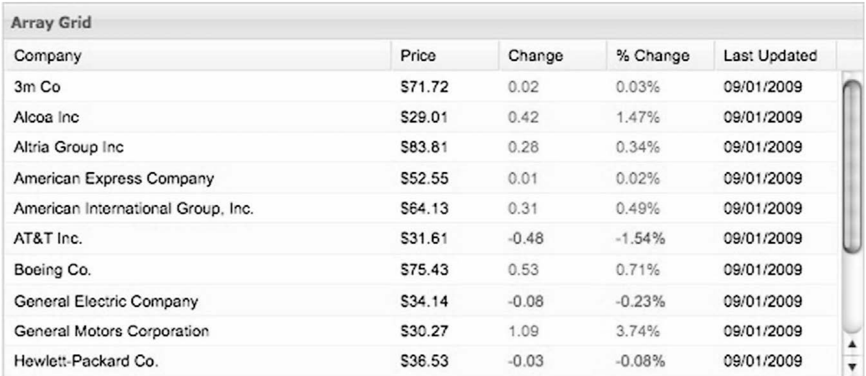
本章将基于前一章的数据存储，围绕网格面板展开介绍。你将从构建一个网格面板开始学习之旅，它的数据存储从本地内存中读取数组数据。在探索完基础知识之后，你将学习一些高级特性，比如分页和滚动条。最后，你将学习如何编辑网格面板中使用了数据存储的数据并与之交互，以及最后一章中涉及的CRUD操作。

在学习过程中的每一步，你都会了解更多关于网格面板及其支持类的内容。在此之前，我们先认识一下网格面板。

8.1 网格面板简介

乍看之下，网格面板看上去就像一个美化了的HTML表格。HTML表格已经多年被用于展示数据。如果你花点儿时间看一个Ext JS网格的示例，就会意识到这不是一个普通的HTML表格。你可以从这里看一个使用数组存储器的网格面板的实现示例: http://mng.bz/HAcK（参见图8-1）。

在这个基于数组的网格示例中，你可以看到这个部件提供的功能远远超过了典型的HTML表格。这其中包括了列管理功能比如排序、重置大小、重排、展示以及隐藏。鼠标事件同样是开箱即用，它允许在鼠标停留在一行的时候高亮显示这一行，甚至点击选择这行。



Array Grid

Company	Price	Change	% Change	Last Updated
3m Co	$71.72	0.02	0.03%	09/01/2009
Alcoa Inc	$29.01	0.42	1.47%	09/01/2009
Altria Group Inc	$83.81	0.28	0.34%	09/01/2009
American Express Company	$52.55	0.01	0.02%	09/01/2009
American International Group, Inc.	$64.13	0.31	0.49%	09/01/2009
AT&T Inc.	$31.61	-0.48	-1.54%	09/01/2009
Boeing Co.	$75.43	0.53	0.71%	09/01/2009
General Electric Company	$34.14	-0.08	-0.23%	09/01/2009
General Motors Corporation	$30.27	1.09	3.74%	09/01/2009
Hewlett-Packard Co.	$36.53	-0.03	-0.08%	09/01/2009

图8-1　可下载的SDK包中exmaples文件夹下的数组网格示例

这个示例同时也展示了网格面板的视图（也叫作网格视图）是如何通过定制化渲染器来定制的，它被应用在了Change和% Change列上。这些定制化的渲染器基于正负数来给文本上颜色。

这个示例仅仅是浮光掠影地过了一下网格面板的配置和扩展能力。要想透彻地理解网格面板以及为什么它的扩展性如此之好，你需要知道更多的支持类。

深入学习

操控网格面板关键的支持类有`grid.view`、`SelectionModel`以及`store`。让我们来快速看一个网格面板的实现，并且看看每个类在网格面板工作的机制中发挥什么样的作用（参见图8-2）。

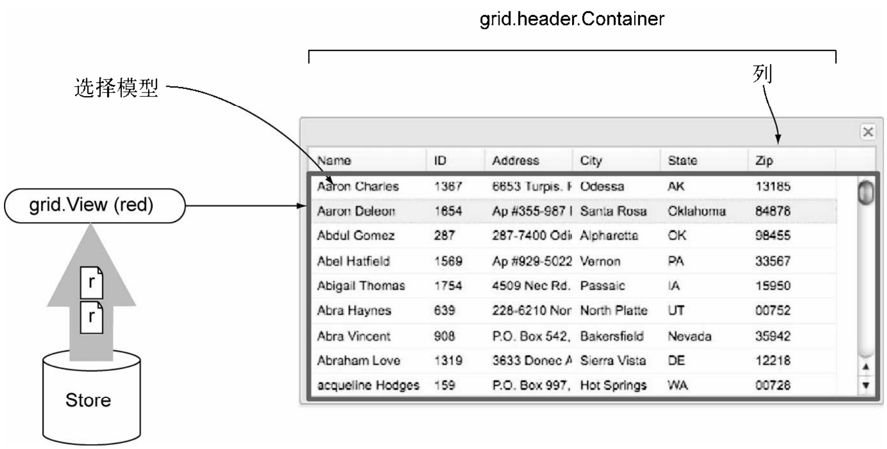


图8-2　网格面板的支持类：`grid.View`、`SelectionModel` 和`Store`

在图8-2中，你能看到一个网格面板和它的支持类被高亮了。从头看起，在数据源中可以看到`Store`类。数据存储用一个读取器来工作，读取器负责从数据源映射数据点，并填充到数据存储中。它们可以分别用数组、XML和JSON读取器来读取数组、XML和JSON数据。当读取器解

析数据时，它被组织成记录，就是这样被组织存放到数据存储中的。

`grid.View`类是网格视图的一个UI组件。它负责读取数据，并控制在屏幕上画出数据。

列是把数据字段从每一个记录中映射数据字段到屏幕上的类。它们通过`dataIndex`属性来做到这一点。`dataIndex`被设置在每一列上，负责显示它从数据字段中映射到的数据。

最后，`SelectionModel`是一个和视图一起工作用来让用户在屏幕上选择一个或多个项目的支持类。Ext JS还现成地支持行、单元、选择框以及树选择模型。`SelectionModel`用来跟踪你在屏幕选择了什么。

现在，你已经学习了基础知识，接下来构建一个简单的网格面板。

8.2 构建一个简单的网格面板

实现网格面板时，你通常需要先配置数据存储。这是因为配置数据列直接关系到配置数据存储的字段。让我们继续使用你在前一章节构建的雇员存储，通过声明渲染网格面板所需要的数据列来连接到网格面板上，如下代码清单所示。

代码清单8-1 创建一个`ArrayStore`，并将之绑定到网格面板

```
var arrayData = [
    ['Jay Garcia',    'MD'],
    ['Aaron Baker',   'VA'],
    ['Susan Smith',   'DC'],
    ['Mary Stein',    'DE'],
    ['Bryan Shanley', 'NJ'],
    ['Nyri Selgado',  'CA']
];

var store = Ext.create('Ext.data.ArrayStore', {
    data       : arrayData,
    fields     : ['fullName', 'state']
});

var grid = Ext.create('Ext.grid.Panel', {
    title        : 'Our first grid',
    renderTo     : Ext.getBody(),
    autoHeight   : true,
    width        : 250,
    store        : store,                    ❶ 引用数据存储
    selType      : 'rowmodel',               ❷ 设置选项模型
    singleSelect : true,
    columns      : [                         ❸ 将数据索引映射到列
        {
            header    : 'Full Name',
            sortable  : true,
            dataIndex : 'fullName'
        },
        {
            header    : 'State',
            dataIndex : 'state'
```

8

```
        }
    ]
});
```

首先要做的是通过name和state值来引用关联的数据存储❶。数据存储中的值会映射到column的值上❸。你需要在存储中为每一列设置相对应的dataIndex。把Full Name列定义为可排序的。最后，指定rowModel为选择类型❷。图8-3展示了面格在屏幕上的样子。

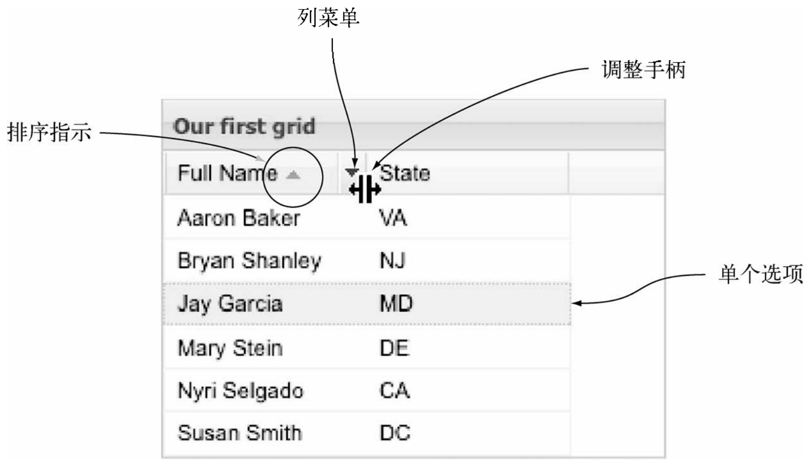


图8-3 第一个展现在屏幕上的网格面板，展示了配置的行SelectionModel以及可排序的Full Name列

可以看到数据并没有按照定义的顺序排列。这是因为在截屏之前，我们点击了Full Name列，这触发了这一列的点击事件处理程序。点击事件处理程序检查这一列是否是可排序的（Full Name是可排序的），如果是，则调用数据存储的sort方法，并传入数据字段（dataIndex），这里是fullName。sort方法继而根据刚刚传入的数据字段来排列所有的数据记录。它先按照升序排列，然后触发降序。在State列上点击则不会触发任何的排序，因为你没有像对Full Name栏做的那样定义sortable:true。

网格面板还有一些其他可以使用的特性。可以拖放列来重新排列，拖动大小重置手柄来重置大小，或者在鼠标移动到一个特定列上的时候点击列出现的菜单图标。

要使用选择模型，需要点击一行来选定它。一旦完成了这步操作，可以在键盘上按向上键和向下键来浏览其他行。可以通过去除singleSelect: true并启用multiSelect: true修改选择模型。重新加载页面可以通过典型的操作系统多选手势比如Shift-click或者Ctrl-click来选择多个数据项。

创建第一个网格面板简直太容易了，不是吗？显而易见的是，除了显示数据和排序，网格面板还有很多其他内容。像分页和为手势（比如右击）设置事件处理程序这样的特性是经常会使用的。这些高级的使用正是我们即将要来学习的。

8.3 高级网格面板构建

前一节构建了一个使用静态内存中数据的网格面板。你实例化了支持类的所有实例，这会帮助你来了解它们。就像框架的很多其他组件一样，网格面板和它的支持类也有其他的配置模式。在构建高级网格面板的过程中，你将会通过几个支持类来了解一些其他模式。

8.3.1 你在构建什么

你将要构建的网格面板会用到一些高级概念，第一个是使用数据存储来查询随机生成的大数据集，并给你使用分页工具条的机会。你将会学习使用TemplateColumn类来给两列数据创建一些定制的渲染器。其中之一将会给ID列上色，另外一个将会更加高级，它会连接地址数据到一列上。

在构建完成这个网格面板之后，你将会再重来一次设置一个rowdblclick处理程序。我们将介绍上下文菜单，正如网格面板的rowcontextmenu事件。如果你有螺旋桨帽子，就带上加把劲儿吧，因为我们将会在这里接触到很多东西！

8.3.2 所需的数据存储和模型

首先必须配置支持的数据存储。你将用到和第7章节相似的雇员模型数据存储定义。本节涵盖这一内容，用来唤醒你的记忆。可以在examples/ch08/datastores.js中找到如下的代码清单里的内容。

8

代码清单8-2 配置数据模型和存储

```
Ext.define('Employee', {
    extend     : 'Ext.data.Model',
    idProperty : 'id',
    fields     : [                                      ❶ 强制数据类型
        {name  : 'id', type : 'int'},               <-
        {name  : 'departmentId', type : 'int' },
        {name  : 'dateHired', type : 'date', format : 'Y-m-d'},
        {name  : 'dateFired', type : 'date', format : 'Y-m-d'},
        {name  : 'dob', type : 'date', format : 'Y-m-d'},
        'firstName',
        'lastName',
        'title',
        'street',
        'city',
        'state',
        'zip'
    ]
});

var urlRoot = 'http://extjsinaction.com/crud.php'
                    + '?model=Employee&method=';

var employeeStore = Ext.create('Ext.data.Store', {
```

```
    model     : 'Employee',
    pageSize : 50,                                    ❷ 为网格面板分页做准备
    proxy     : {
        type    : 'jsonp',
        api     : {
            create  : urlRoot + 'CREATE',
            read    : urlRoot + 'READ',
            update  : urlRoot + 'UPDATE',
            destroy : urlRoot + 'DESTROY'
        },
        reader : {
            type              : 'json',
            metaProperty      : 'meta',
            root              : 'data',
            idProperty        : 'id',
            totalProperty     : 'meta.total',
            successProperty : 'meta.success'
        },
        writer : {
            type             : 'json',
            encode           : true,
            writeAllFields : true,
            root             : 'data',
            allowSingle      : true,
            batch            : false,
            writeRecords     : function(request, data) {
                request.jsonData = data;
                return request;
            }
        }
    }
});
```

代码清单8-2中的内容应该会让你感觉似曾相识吧。不同点在于强制了模型的数据类型❶，并且给存储配置注入了`pageSize`属性❷。`pageSize`属性使得与分页工具条的整合变得容易起来，你将会在本章后面看到这一点。

在我们继续之前，请注意一点，正在用`employeeStore`引用指向一个数据存储的实例。你将会在配置网格的时候用到这个引用。

8.3.3　创建列

你在第一个网格面板中用到的列都很无趣，它们不过是映射列到记录的数据字段上去。在这个示例中，你将会使用下面代码清单所示的两个模板列，其中一个允许使用地址数据字段来构建复合的、风格化的单元。模板列仅仅是一个可用的选择（你还可以使用响应、布尔和数字列）。

代码清单8-3　创建列

```
var columns = [
  {                                              ❶ 使用模板列
      xtype     : 'templatecolumn',
```

```
        header    : 'ID',
        dataIndex : 'id',
        sortable  : true,
        width     : 50,
        resizable : false,
        hidden    : true,                                              ❷ 隐藏ID列
        tpl       : '<span style="color: #0000FF;">{id}</span>'        ❸ 将ID渲染为蓝色
    },
    {
        Header    : 'Last Name',
        dataIndex : 'lastName',
        sortable  : true,
        hideable  : false,
        width     : 100
    },
    {
        header    : 'First Name',
        dataIndex : 'firstName',
        sortable  : true,
        hideable  : false,
        width     : 100
    },
    {
        header    : 'Address',
        dataIndex : 'street',
        sortable  : false,
        flex      : 1,
        tpl       : '{street}<br />{city} {state}, {zip}'              ❹ 为地址设置模板
    }
];
```


配置这些列和配置前面网格上的列是十分相似的。即便如此，还是有一些显而易见的不同之处。第一，使用了模板列❶来把ID列渲染成了蓝色❸。请注意ID列默认是隐藏的❷。并且，把Last Name和First Name列的hideable属性设置成了false，这会阻止它们通过列菜单隐藏掉。你会在渲染了这个网格面板之后看到这一点是如何起作用的。

Address列则有点特殊：被禁止排序。这么做是因为是基于记录中的其他数据字段如city、state和zip的组合来渲染这一列的。它是通过地址模板来完成这一操作的❹。允许在这其中每一列上做排序。

现在已经构建了一个列配置对象的数组了，让我们继续拼凑分页网格面板吧！

8.3.4 配置高级网格面板

现在你刚刚了解了所有需要用来配置分页网格面板的内容。你将需要配置分页工具条，这将会被用在网格面板底部的工具条上，如下面的代码清单所示。

代码清单8-4 配置高级网格面板

```
var pagingToolbar = {                                ❶ 配置分页工具条
    xtype       : 'pagingtoolbar',
```

```
        store       : employeeStore,
        dock        : 'bottom',
        displayInfo : true
    };

    var grid = Ext.create('Ext.grid.Panel', {        ❷ 创建网格面板
        xtype         : 'grid',
        columns       : columns,
        store         : employeeStore,
        loadMask      : true,
        selType       : 'rowmodel',
        singleSelect  : true,
        stripeRows    : true,
        dockedItems   : [
            pagingToolbar
        ]
    };
```

在代码清单8-4中，使用XType快捷地配置分页工具条和网格面板。对于分页工具条配置❶，绑定了早先配置过的雇员数据存储并把pageSize属性设置成50。这么做使得分页工具条绑定到了数据存储上，并且使之得以控制请求。pageSize属性会被发送到远端的服务器上作为limit属性，它会保障数据存储每次请求接受50（或更少）条记录。分页工具条会使用limit属性和服务器端返回的totalCount属性一起计算数据集一共有多少“页”。最后一个配置的属性displayInfo，它指示分页工具条去显示一个小的文本块，用来展示当前的页面位置，以及还有多少记录可以通过翻页获得（记得totalCount）。我们将会在你渲染网格面板的时候指出这一点。

然后要配置一个网格面板实例❷。在这个实例中，绑定了配置变量columns、employeeStore和pagingToolbar。loadMask属性被设置成true，这会指示网格面板去创建一个Ext.LoadMask的实例并把它绑定到bwrap（body wrap）元素上，该元素是一个最终会包含面板标题栏下所有元素的标签。这些元素包括有顶端的工具条、内容体、底部的工具条，以及fbar（底部按钮页脚栏）。LoadMask类绑定到很多存储器发布的事件上，存储器会根据自身状态来选择显示或隐藏。比如说，当存储器实例化一个请求，它会罩住bwrap元素；而当请求完成的时候，它使之显示。

为pagingToolbar的XType配置对象设置dockedItems属性，这会在网格面板的底部工具条上渲染一个带有配置数据的分页工具条部件的实例。

网格面板现在已经配置好了，并且准备好放入到一个容器中并渲染出来了。可以把网格面板渲染到一个文档体元素中，但是让我们先把它放作一个Ext.Window实例的子节点吧。这样，可以轻易地重设网格面板的大小，并且看到像自动调整Address列大小的特性是怎么工作的。

8.3.5　给网格面板配置一个容器

让我们给高级网格面板配置一个容器。一旦渲染了容器，就会初始化刚刚创建的查询以获取

远端的数据存储：

```
Ext.create('Ext.Window', {
    height : 350,
    width  : 550,
    border : false,
    layout : 'fit',
    items  : grid
}).show();

employeeStore.load();
```

在这里执行了两步操作。第一步是创建`Ext.Window`，它使用了Fit布局，并且拥有一个网格面板作为其唯一的项。使用了方法链直接在构造器调用的结果上调用`show`方法。然后，对`employeeStore`执行`load`操作。渲染了的网格面板如图8-4所示。从你的劳动成果上可以看到，网格面板在整齐的一列上显示了一个复合的地址字段，它是动态规划大小的并且不能排序，与此同时，所有其他有固定初始大小的列是可以排序的。

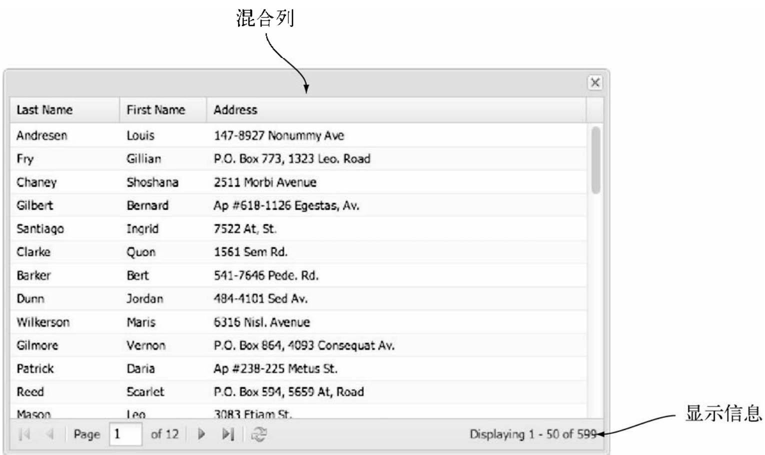


图8-4　高级网格面板的实现结果

通过Firebug工具快速看一眼第一个请求的通信，可以看到被送往服务器端的参数。图8-5展示了这些参数。

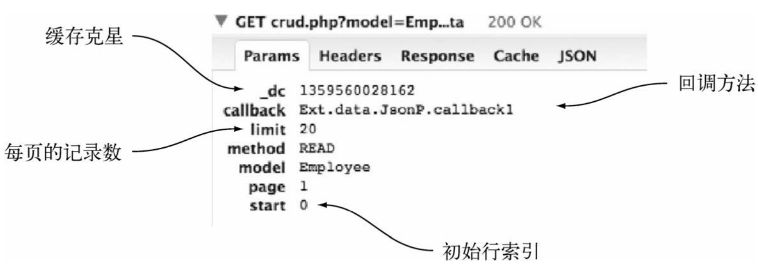


图8-5　发送到远端服务器上的请求分页数据的参数

我们刚刚已经在讨论分页工具条的过程中说过了callback、limit和start参数，但是还没有看到过_dc参数。_dc参数被认作是“缓存克星”参数，它对于每个请求是唯一的，并包含了Unix epoch格式的时间戳信息（从计算机时间，也就是1970年1月1日12时开始的秒数）。因为这个数值对于每一个请求是唯一的，这个请求绕过代理，并且阻止它们被拦截而返回缓存数据。

还没有看到ID列，这是因为把它配置为隐藏列。要启用它，可以使用列菜单，并选上ID列，如图8-6所示。

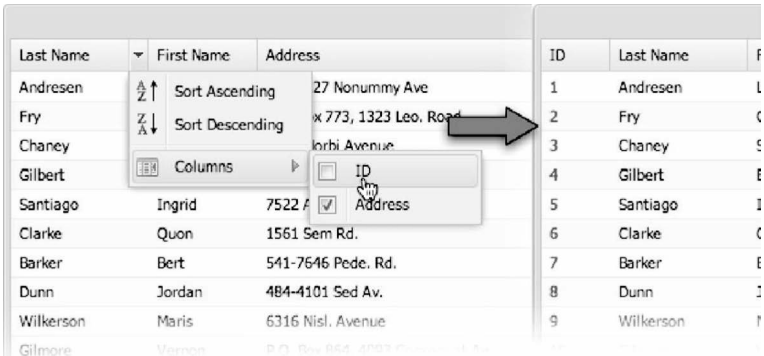


图8-6　通过列菜单启用ID列

在列菜单上选上ID列之后，就会看到它出现在网格视图里面了。在这个菜单中，还可以指定某一列被排序的方向。看了列菜单之后，可能还会注意到一件事情，就是First Name和Last Name的菜单选项不见了。这是因为设置了hideable标志为false，这阻止了它们对应的菜单项渲染出来。列菜单也是按照需求排序列的很好途径。

现在已经把网格面板构建好了，可以为它配置一些事件处理程序，以便与之可以进行更多的交互。在此之前，让我们探索一下使用分页工具条外的另一个选择：缓冲滚动分页。

8.3.6　缓冲滚动分页

想要避免呈现分页工具条，而不必等待浏览器检索整个数据集吗？你真幸运！Ext JS 4.1为网格引入了网格滚动分页。这一新的网格滚动分页允许Ext JS在场景背后分页并在显示数据之前检索数据。我们知道这听上去有点复杂，所以让我们来实际地探索一下吧。

缓冲滚动分页背后的主体思想依然是把大数据集分拆到多个页面中。在这一系统中，你在用户滚动到底或顶之前预读取数据。比如，如果用户已经滚动到接近网格底部了，就在用户滚到底之前检索一定量的数据。这样，就避免了让用户在应用于服务器端交互并渲染结果的时候等待了。要启动这一方案，需要配置一个缓冲数据存储，如下面的代码清单所示。

代码清单8-5　支持缓冲的雇员数据存储

```
var url = 'http://extjsinaction.com/crud.php?model=Employee&method=READ';
var bufferedEmployeeStore = Ext.create("Ext.data.Store", {
    model      : 'Employee',
```

```
    pageSize   : 50,
    buffered   : true,
    remoteSort : true,
    sorters    : {
        property  : 'lastName',
        direction : 'ASC'
    },
    proxy      : {
        type   : 'jsonp',
        url    : url,
        reader : {
            type            : 'json',
            root            : 'data',
            idProperty      : 'id',
            successProperty : 'meta.success',
            totalProperty   : 'meta.total'
        }
    }
});
```

❶ 确定页面的大小

❷ 将buffered设为true

❸ 读取查询结果的总数量

首先需要标识出页面的大小❶。需要调整这个大小以获取最佳的后台服务性能以及最小的UI延迟。把网络速度和传输的字节数量也考虑进去。

同时记得把buffered属性设置为true❷。这么做会启用缓冲，并允许缓冲滚动分页工作。应该在属性上排序，这样可以在屏幕上比较容易地跟踪滚动进度。在这个示例中，我们在lastName上启用排序功能。一个好的经验法则是在渲染到网格面板上的第一列上进行排序。

最后，记住要从服务器端读取查询结果的总数量❸。如果不这么设置，数据存储就不能够计算如何配置网格面板的滚动条。

8

对于缓冲滚动分页，需要给网格配置一个垂直的滚动条。要实现它，使用前面定义的列和Window实例来渲染网格面板。正如将要看到的，最大的不同是给网格面板添加了一个verticalScroller：

```
var grid = Ext.create('Ext.grid.Panel', {
    xtype            : 'grid',
    columns          : columns,
    store            : bufferedEmployeeStore,
    loadMask         : true,
    verticalScroller : {
        trailingBufferZone : 10,
        leadingBufferZone  : 10
    }
});
```

trailingBufferZone使得可以配置应该被存在内存中的“上面”（在离开可视的屏幕区域之后，在用户向下滚动之后）的行数。leadingBufferZone使得配置应该在查询之间被检索出来的行数。

看一看代码给浏览器带来的结果上的改变。你也像我们一样被震撼了吗？记住只有在合适的时候使用这一特性。如果打算在网格上启用编辑功能，请注意多项目的编辑用工具条分页做起来更方便。网格上的静态分页使得跟踪用户交互更容易。所以在引入很棒的特性之前要仔细地计划

一下。

让我们开始探索如何应用事件处理程序以允许用户与网格进行交互。

8.3.7　为交互应用事件处理程序

要创建基于行的用户交互，需要绑定事件处理程序到网格面板发布的事件上去。这里你将会学习使用rowdblclick事件在侦测到行上的双击事件时展示对话框。同样地，将使用鼠标坐标来监听一个contextmenu（右击）事件，以创建并展示单个项目的上下文菜单。

将从创建一个方法用来格式化Ext JS警告信息框里面的信息开始，然后继续创建一个特定的事件处理程序，如下面的代码清单所示。可以把这段代码插到配置网格面板之前的任何地方。

代码清单8-6　为数据网格创建事件处理程序

```
var doMsgBoxAlert = function() {
    var record    = grid.selModel.getSelection()[0];
    var firstName = record.get('firstName');
    var lastName  = record.get('lastName');
    var msg = String.format('The record you chose:<br /> {0}, {1}',
           lastName , firstName);
    Ext.MessageBox.alert('', msg);
};
var doRowDblClick = function() {
    doMsgBoxAlert();
};
var doRowCtxMenu = function(view, record, item, index, e) {
    e.stopEvent();
    if (!view.rowCtxMenu) {
        view.rowCtxMenu = Ext.create('Ext.menu.Menu', {
            items : {
                text     : 'View Record',
                handler : function() {
                   doMsgBoxAlert();
                }
            }
        });
    }
    view.rowCtxMenu.showAt(evtObj.getXY());
};
```

❶ 显示警示信息栏
❷ 添加rowdlclick处理程序
❸ 添加项目内容菜单处理程序
❹ 隐藏浏览器内容菜单
❺ 创建静态菜单实例

在代码清单8-6中，创建了三个方法。第一个是doMsgBoxAlert❶，它是指示网格面板生成事件的工具类。它使用了SelectionModel的getSelection方法来获取被选中记录的引用，并使用record.get方法来抽取姓和名的数据字段。它使用这些数据字段来显示一个包含了这两个属性的警告信息框。

上下文菜单典型选择项

大多数桌面应用都在用户右击时选中某项目。因为Ext JS并没有在网格面板上提供开箱即用的这一功能，你可以强制选中用户右击的项目。这样做可以使应用更像桌面应用。

接下来创建第一个处理程序：doRowDblClick❷。这一方法的全部工作就是执行我们刚讨论过的doMsgBoxAlert方法。

最后一个方法是doRowCtxMenu❸，它更加复杂，接受5个参数。第一个是网格视图的引用，第二个是数据存储中被操作的数据记录，第三个是项目元素，第四个是项目的索引，以及第五个是Ext.EventObject的实例。知道这些是很重要的，因为在某些浏览器上，比如Mac OS X的Firefox，需要阻止浏览自己的上下文菜单显示出来。这就是为什么要把调用e.stopEvent❹作为第一件事情。调用stopEvent阻止了浏览器原生的上下文菜单显示。

接下来，处理程序使用record参数来强制选择通过SelectionModel的select方法选中的行。它是通过record参数传入的。

然后检测下网格是否有rowCtxMenu属性，它的第一个执行是false，解释器就是进入这个代码分支。这么做是因为要在它不存在的时候创建一个菜单。如果逻辑中没有这个分叉，每次上下文菜单被调用时就需要重建菜单，这太浪费时间了。

然后把grid上的rowCtxMenu属性❺赋值为一个Ext.menu.Menu新实例的结果。菜单项的第一个属性是菜单项显示时展示的文本内容。另一个内联定义的处理程序方法会导致doMsgBoxAlert在引用的记录上被调用。

最后一点代码是在新创建的rowCtxMenu上调用showAt方法，它需要*X*坐标和*Y*坐标来展示菜单。通过直接传入evtObj.getXY()的结果给showAt方法来做到这一点。evtObj.getXY会在事件发生的时候直接返回精确的坐标信息。

事件处理程序现在武装完毕，并准备好被调用了。在可以在网格中使用之前，需要配置它们为监听器，如下所示：


```
listeners : {
    itemcontextmenu : doRowCtxMenu,
    itemdblclick    : doMsgBoxAlert
}
```

要给网格配置事件处理程序，需要添加一个listeners配置对象，并映射到要处理的事件上。因为事件处理程序只能处理一行记录，所以必须要强制单行选择。可以通过添加rowmodel的selType，并把singleSelect设置为true。

刷新页面，并在网格上触发一些双击和右击事件。发生什么了？参看图8-7。

现在双击任何记录都会导致Ext JS警告信息框出现。同样地，右击一行数据会导致一个定制化的上下文菜单出现。如果点击View Record菜单项，一个警告框就会出现。

给网格添加用户交互就是这样的简单。高效开发UI交互的一个关键就是只在需要的时候才实例化并渲染部件一次，就像对上下文菜单做的那样。虽然这一技巧对于阻止重复项很好用，但是到清除的时候就显得不合格了。还记得组件生命周期的销毁部分把？可以在监听器列表上附加一个destroy处理程序方法来在网格面板销毁的时候销毁上下文菜单：

```
listeners       : {
    itemdblclick   : doRowDblClick,
```

```
        itemcontextmenu : doRowCtxMenu,
        destroy         : function(grid) {
            if (grid.rowCtxMenu) {
                grid.rowCtxMenu.destroy();
            }
        }
    }
```

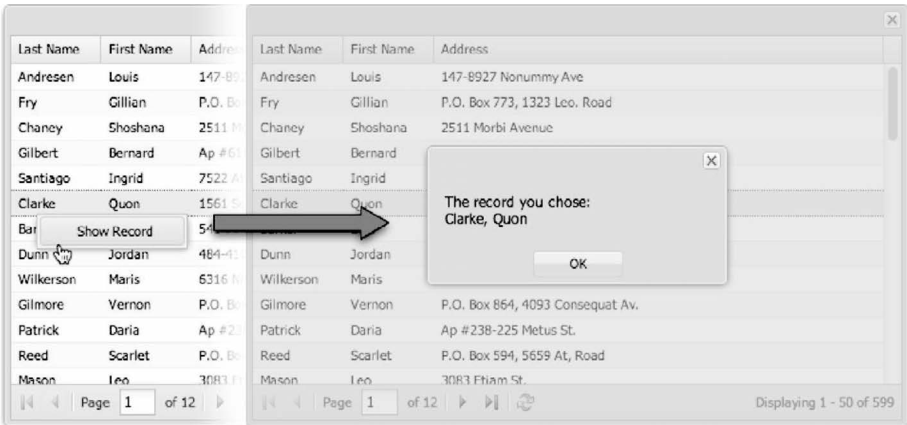


图8-7　给高级网格添加上下文菜单处理程序的结果

在这一代码片段中，内联添加了destroy事件处理程序，而不是给它创建单个引用方法。destroy事件永远传入发布事件的组件，也就是标记的grid。在这个方法中，要检测rowCtxMenu变量是否存在。如果这个项存在，就要调用它的destroy方法。

上下文菜单的清理是开发人员经常容易遗漏的地方之一，它会产生大量多余的DOM节点垃圾，那样会消耗大量内存，导致应用程序的性能在一段时间之后下降。如果给组件附加上下文菜单，一定要记住注册destroy事件处理程序来销毁一切存在的上下文菜单。

这样你就拥有它啦！现在你知道了如何操作一个网格面板。接下来我们要探索如何构建一个可用的可编辑的网格面板，它可以内联修改数据，就像在Excel这样的桌面电子表格制作应用里面操作一样。你将通过使用Ext JS 4新引入的网格面板编辑插件来做到这一点。

8.4　在网格面板上编辑数据

在Ext JS早期的版本中，必须要实现一个独立的可编辑的网格面板来允许在网格上编辑内容。有了Ext JS 4之后，可以重用已有的网格面板并通过插件来启用编辑功能。编辑插件给网格面板注入了编辑的能力，这样就可以重用网格的数据存储来持久化数据了。可以选择使用RowEditing或者CellEditing插件，这取决于想要在行级别还是单元格级别启用编辑功能。

使用网格面板的编辑插件可以大大简化开发过程，但是还是需要安装事件处理程序来对数据存储进行CRUD操作。我们将通过扩展的复杂网格面板来探索学习这一点。但需要一些改变才能

使用RowEditing插件处理数据。

已经很兴奋了吧？让我们先睹为快，在继续之前看看有了启用了RowEditing插件的网格面板是什么样的（参见图8-8）。

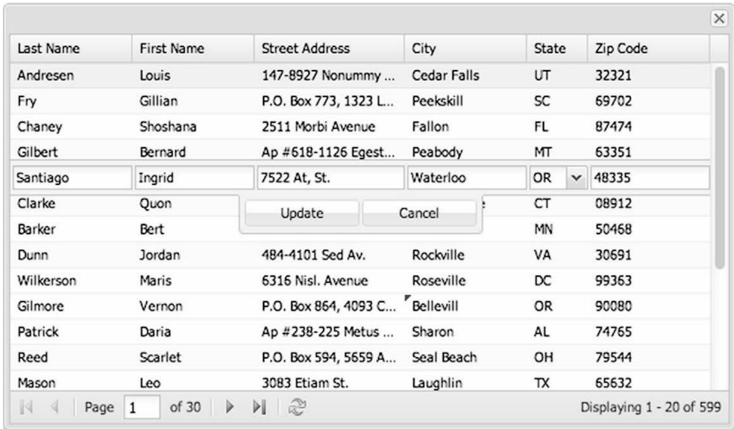


图8-8 先睹为快下你在本节将要构建什么

从启用编辑插件开始，这样便可以了解到实现编辑的。然后我们就讨论创建UI部件用于支持添加和删除的CRUD操作的方方面面，以及如何从存储器中获取修改过的数据记录，或者甚至用我们在前一章节介绍过的数据存储来拒绝修改。构建一个不存储数据的可编辑网格面板是没有用处的，所以我们将趁此机会给你展示一下如何编码CRUD操作。这将会是你到目前为止看到过的最复杂的代码，我们将一步步地创建它们。

第一步是启用RowEditing插件，并让编辑器工作起来。然后要回过头来逐个地添加所有的CRUD操作。

8.4.1 启用编辑插件

因为要扩展先前创建好的复杂网格面板，你会看到一些相同的代码和模式。我们这么做可以让代码的流程变得尽可能简洁流畅。大多数情形下代码会有一些变化，所以请花时间读每一个字。我们会指出所有相关的修改。

本节基于第7章介绍的数据存储来展开。你将会从下面的代码清单开始，创建这个可编辑网格的支持类的实例，如数据存储、RowEditing插件以及列编辑器配置。

代码清单8-7 创建网格面板编辑需要的支持类的实例

```
Ext.define('State', {
    extend : 'Ext.data.Model',
    fields : ['id', 'state']
});

var url = 'http://extjsinaction.com/crud.php??model=State&method=READ';
```

```
var stateStore = Ext.create("Ext.data.Store", {          <-- ❶ 创建数据存储
    model : 'State',
    proxy : {
        type   : 'jsonp',
        url    : url,
        reader : {
            type            : 'json',
            root            : 'data',
            idProperty      : 'id',
            successProperty : 'meta.success'
        }
    }
});
var rowEditing = Ext.create('Ext.grid.plugin.RowEditing', {   <-- ❷ 设置新的RowEditing实例
    clicksToEdit : 2,
    autoCancel   : false
});
                                                  ❸ 添加文本编辑器配置对象
var textField = {                               <--
    xtype : 'textfield'
};

var stateEditor = {                               <-- ❹ 设置组合框配置对象
    xtype         : 'combo',
    triggerAction : 'all',
    displayField  : 'state',
    valueField    : 'state',
    store         : stateStore
};
```

从创建用于修改数据记录State字段的组合框编辑器❹的数据存储❶开始，然后创建一个RowEditing的网格插件实例❷。把clicksToEdit设置为2，这样双击事件就会带起一个编辑器，这和大多数用户交互的流程是一样的。autoCancel被设置为false，这样当用户决定编辑另一行的时候就会保持更改。我们喜欢把它设置为false是因为用户总是可以通过RowEditing插件内置的取消按钮取消变化。

接下来创建一个可重用的单行文本框配置对象，它会被用于差不多是可编辑网格面板❸的每一列上。最后声明一个组合框配置对象❹。它会被用于网格面板的State列上。

现在可以声明列并赋值编辑器了，正如下面的代码清单所示。

代码清单8-8　创建带编辑器的列

```
var columns = [
   {
      Header    : 'Last Name',
      dataIndex : 'lastName',
      sortable  : true,
      editor    : textField          <-- ❶ 使用文本字段编辑器
   },
   {
      header    : 'First Name',
      dataIndex : 'firstName',
```

```
        sortable  : true,
        editor    : textField
    },
    {
        header    : 'Street Address',
        dataIndex : 'street',
        flex      : 1,
        sortable  : true,
        editor    : textField
    },
    {
        header    : 'City',
        dataIndex : 'city',
        sortable  : true,
        editor    : textField
    },
    {
        header    : 'State',
        dataIndex : 'state',
        sortable  : true,
        width     : 50,                          ❷ 指定州编辑器
        editor    : stateEditor         <┘
    },
    {
        header    : 'Zip Code',
        dataIndex : 'zip',
        sortable  : true,
        editor    : textField
    }
];
```

8

当复查columns配置数组，会看到一些熟悉的属性，比如header、dataIndex和sortable。还可以看到代码块里面的一个新成员editor❶，它可以给每一列定义特定的编辑器。你还会注意到给State列定义的stateEditor❷。

现在已经配置好了存储器、编辑器和列，可以像下面的代码清单一样创建分页工具条了。对于这个任务，要重用早先创建好的employeeStore实例了。

代码清单8-9 创建分页工具条、网格面板和窗口

```
var pagingToolbar = {
   xtype        : 'pagingtoolbar',
   store        : employeeStore,
   displayInfo  : true
};

var grid = Ext.create('Ext.grid.Panel', {
   columns      : columns,
   store        : employeeStore,
   loadMask     : true,
   bbar         : pagingToolbar,                 ❶ 启用RowEditing插件
   plugins      : [ rowEditing ],       <┘
   stripeRows   : true,
```

```
    selType       : 'rowmodel',
    viewConfig    : {
        forceFit : true
    },
    listeners     : {
        itemcontextmenu : doRowCtxMenu,
        destroy         : function(thisGrid) {
            if (thisGrid.rowCtxMenu) {
                thisGrid.rowCtxMenu.destroy();
            }
        }
    }
});
 Ext.create('Ext.Window', {
     height       : 350,
     width        : 600,
     border       : false,
     layout       : 'fit',
     items        : grid
}).show();

employeeStore.load();
```

在代码清单8-9中，创建了网格面板剩下的部分，以使用雇员存储的分页工具条开始。接下来用columns、employeeStore和pagingToolbar创建了网格面板❶。

然后创建了网格面板的容器，也就是Ext.Window的一个实例，并把它的布局设置为'fit'。在窗口刚实例化完毕就用链式方法调用来展示它。

最后，调用employeeStore.load❷并传入指定了参数的配置对象到服务器端去。这保障了从记录0开始并限制返回的记录数为50。

所有这一步骤需要的拼图都已经完成了。现在可以渲染网格面板并开始编辑数据了。图8-9展示了带有RowEditing插件的网格面板。

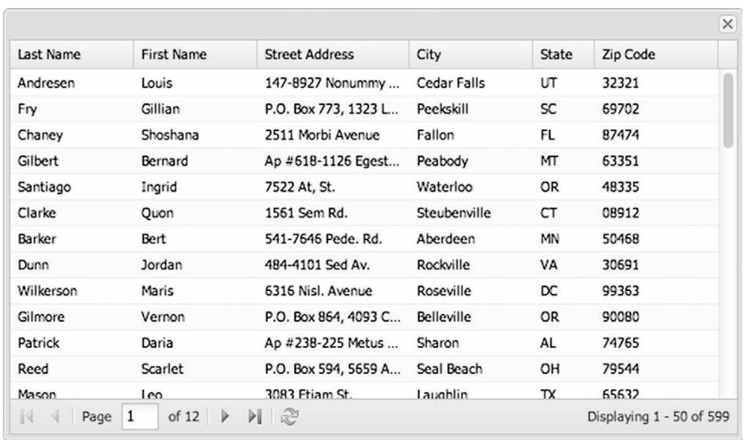


图8-9 可编辑的网格面板

可以看到渲染好的可编辑的网格面板、分页工具条，还有等待被修改的数据。刚开始看上去，它就像是一个普通的网格面板，但是在表象下面它是一个等待被发掘的全新功能。我们将花点儿时间探讨一下该怎么使用它。

8.4.2　浏览一下你的可编辑网格面板

可以使用鼠标或者键盘动作来浏览单元格，并进入或者退出编辑模式。可以用鼠标来开始编辑模式，双击一个单元格，然后编辑器就会显示出来，正如前面的图8-8所示。然后可以修改数据，点击或双击另外一个单元格，或者页面上的任何地方来让编辑器消失掉。重复这一过程可以更新任意多的单元格。

可以通过添加属性`clicksToEdit`到RowEditing插件上并指定一个整数值来控制要点击多少下才可以编辑一个单元格。一些应用允许通过单击单元格来编辑，如果想要这样的话，只需要把`clicksToEdit`设置成1就可以了。

作为命令行爱好者，我们觉得键盘导航提供了比鼠标更强大的能力。如果你是一个Excel或者类似的电子表格应用程序的高级用户，就知道我们在说什么了。要开始用键盘导航，可以用鼠标先聚焦到想要编辑的第一个单元格上。这会立即把焦点放到你需要的地方去。可以用Tab键，或者Shift-Tab组合键，来往左或右移动。还可以使用方向键来聚焦任意的单元格。

要用键盘进入编辑模式，按下回车键，这会显示一个该单元格的编辑器。在编辑模式下，可以通过按下Tab键向右移动一个单元格，或者Shift + Tab组合键向左移动一个单元格来编辑相邻的单元格。

要退出编辑模式，可以再次按下回车键或者按下Esc键。如果输入的或修改的数据验证正确，这条记录就会被修改，数据字段就会被标记成脏数据（修改过的）。可以在图8-10中看到不少被修改过的数据字段。

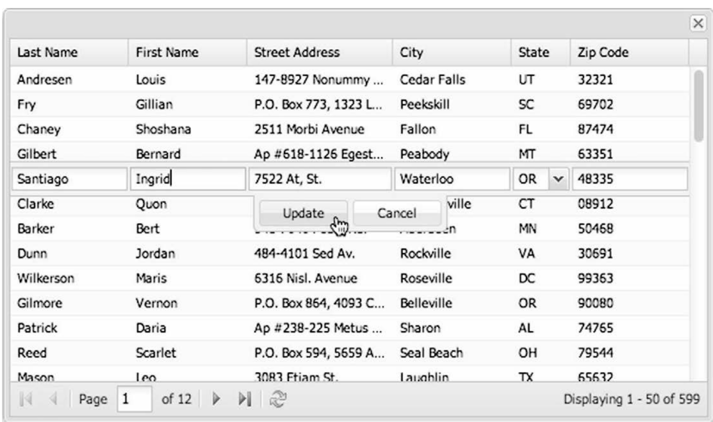


图8-10　带有编辑器的网格面板，其中有脏数据字段的标记

当退出编辑器，有赖于数据字段是否有验证以及验证的结果，数据会被丢弃。要检验这点，

编辑邮政编码的单元格，并输入多于或少于5位整数，然后通过按下回车键或者Esc键退出编辑模式。

现在可以编辑数据，但是假如不保存修改的话，编辑是没有用的。这时候就需要进入下一个构建环节了：增加CRUD层。

8.5　加入 CRUD

通过网格面板，服务器端的CRUD请求可以被自动或者手动触发。自动的请求发生于客户端超时的时候一个记录被修改或者有修改发生，此时便会触发一个到服务器的请求。要创建自动的CRUD，需要创建自己的逻辑来发送请求，也可以用前一章创建的雇员存储来轻松做到这一点，这正是本章后面将要采用的办法。

8.5.1　添加保存和拒绝逻辑

从创建保存和拒绝修改方法开始，它会被绑定到按钮放到分页工具条上，正如下面的代码清单所示。保存修改和拒绝修改按钮可以直接用分页工具条来实现。

代码清单8-10　重新配置分页工具条以增加保存和拒绝按钮

```
var pagingToolbar = {
     xtype       : 'pagingtoolbar',
     store       : employeeStore,
     pageSize    : 50,
     displayInfo : true,
     items       : [
          '-',                                    // ❶ 添加间隔符
          {
               text    : 'Save Changes',
               handler : function () {
                    employeeStore.sync();         // ❷ 保存数据
               }
          },
          {
               text    : 'Reject Changes',
               handler : function () {
                    employeeStore.rejectChanges();   // ❸ 拒绝更改
               }
          },
          '-'
     ]
};
```

在代码清单8-10中配置了一个XType为`pagingtoolbar`的配置对象来放入`items`❶。你看到的连字号（-）字符串是`Ext.Toolbar.Separator`的缩写，这会在工具条子项目之间放入一个小的竖条。这么做是因为想要在按钮和常用的分页工具条导航项目之间有一个分隔。用`employeeStore`的❷`sync`操作来实现保存修改，而用`employeeStore`的`rejectChanges`操作

来实现拒绝修改。

常见的`Ext.Toolbar.Button`实例生成的对象也在链表之中。图8-11显示了保存修改和拒绝修改按钮，它们设置有各自的处理程序。正如从图中可以看到的，这两个按钮被整齐地放在了分页工具条的中间，通常情况下这里应该是空的，而且它们被整齐的按钮分隔符分开了。现在可以开始编辑数据，使用装点过的分页工具条上的功能了，以及新创建的CRUD方法。



图8-11 添加了保存修改和拒绝修改按钮的网格面板

8.5.2 保存和拒绝修改

要使用保存和拒绝修改按钮，首先要修改数据。用已经知道的网格面板的功能修改一些数据，然后点击保存修改按钮。应该可以看到网格面板的元素遮盖出现了片刻，然后在保存结束之后消失了。此时，被标记成脏数据的单元格又被标记成干净，也就是提交了。图8-12显示了遮盖动作。

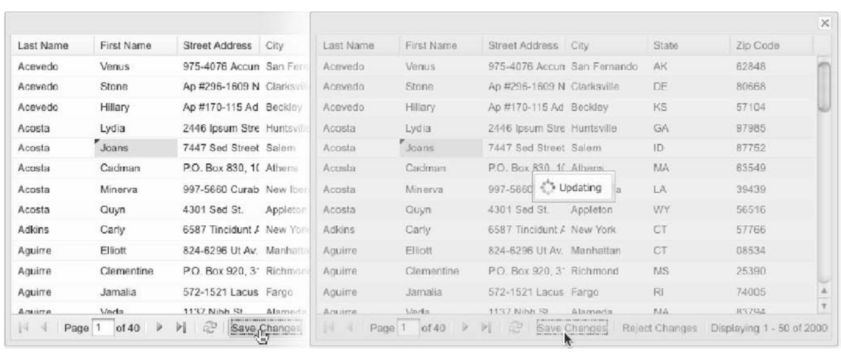


图8-12 当保存请求正在被发送到服务器端时出现的加载遮盖

现在已经看到了执行远端保存和修改数据记录需要什么了，接下来给网格面板添加创建和删除功能，这样才能完成CRUD操作。

8.5.3 添加创建和删除功能

当给保存和拒绝功能配置UI的时候，在分页工具条上添加按钮。虽然也可以用一样的方式添加创建和删除功能，但是最好是使用上下文菜单，因为这使得删除和添加操作更顺畅。想想看，如果你曾经用过一个电子表格应用程序，就会知道右击一个单元格就会显示一个上下文菜单，这其中就有插入和删除的菜单项。我们在这里将会采用同样的范式。

正如对前面添加的功能所做的一样，在构建和配置UI组件之前首先开发如下面代码清单所示的支持方法。我们将增加复杂度。

代码清单8-11 构建删除和插入记录方法

```
var onDelete = function() {                              ❶ 添加删除操作方法
   var selected = grid.selModel.getSelection();
   Ext.MessageBox.confirm(
       'Confirm delete',
       'Are you sure?',
       function(btn) {
           if (btn == 'yes') {
               grid.store.remove(selected);               ❷ 从存储中删除项
               grid.store.sync();
           }
       }
   );
};
```

这里从声明用于删除记录的`onDelete`函数开始❶。通过选择模型找到要删除的记录之后，通过调用`Ext.MessageBox.confirm`要求用户确认是否删除记录。如果得到确认，就通过调用存储上的`remove`❷方法来删除选择的数据记录。

在部署`delete`之前，要添加一个`insert`处理程序。这个程序相对较小：

```
var onInsertRecord = function() {
      var selected = grid.selModel.getSelection();
      rowEditing.cancelEdit();
      var newEmployee      = Ext.create("Employee");
      employeeStore.insert(selected[0].index, newEmployee);
      rowEditing.startEdit(selected[0].index,0);
    };
```

这个方法的目的是定位被右击的行索引，然后插入一行影子记录。这里阐释下它是怎么工作的。

首先通过调用`Ext.create ("Employee")`来创建一条新的数据记录。接下来的调用是`employeeStore`的`insert`方法，它需要两个参数。第一个参数是希望插入记录的索引，第二个则是一个实际记录的引用。这模仿了我们讨论过的电子表格程序，高效地在右击的行之上插入一条数据记录。

最后，要立即开始编辑这条记录。要通过调用RowEditing插件的`startEdit`方法来做到这一点，传入插入新记录的行，并传入`0`（它代表了第一列）。

这就是创建和删除需要的支持方法了。现在可以继续创建上下文菜单处理程序，并重新配置网格来监听`itemcontextmenu`事件了，如下面的代码清单所示。

代码清单8-12　创建网格面板的上下文菜单处理程序

```
var doRowCtxMenu = function(view, record, item, index, e) {    ❶ 添加监听器
 e.stopEvent();

 if (!grid.rowCtxMenu) {                                         ❷ 测试rowCtxMenu是否存在
   grid.rowCtxMenu = Ext.create('Ext.menu.Menu', {
       items : [
                 {
                     text    : 'Insert Record',
                     handler : onInsertRecord
                 },
                 {
                     text    : 'Delete Record',
                     handler : onDelete
                 }
             ]
         });
   }                                                             ❸ 选择被右击的单元格
   grid.selModel.select(record);
   grid.rowCtxMenu.showAt(e.getXY());
};
```

代码清单8-12包含了`doRowCtxMenu`❶，它是网格面板上处理`itemcontextmenu`事件的方法，负责创建和展示有插入和删除操作的上下文菜单。下面说说它是怎么工作的。

`doRowCtxMenu`接受`itemcontextmenu`处理程序传入的5个参数：

- ❑ `view`是一个网格面板上触发事件的视图引用；
- ❑ `record`记录；
- ❑ `item`和`index`，它们标识了被编辑的记录；
- ❑ `e`，它是`Ext.EventObject`的一个实例。

这个方法执行的第一个功能是通过调用`e.stopEvent`来阻止向上冒泡的其他右击事件，这里是阻止浏览器显示自身的上下文菜单。如果不阻止这个事件冒泡，就会看到浏览器的上下文菜单覆盖在你自己的菜单上面，那将会傻而无用。

然后，`doRowCtxMenu`检验❷网格面板是否已经有`rowCtxMenu`属性，然后创建一个`Ext.menu.Menu`的实例，并把它的引用作为`rowCtxMenu`属性存到网格面板上。这非常高效地创建了一个单一菜单，比每次事件触发都创建一个新的`Ext.menu.Menu`实例要高效得多。它会一直存在直到网格面板被销毁，正如你接下来看到的。

传入一个配置对象给`Ext.menu.Menu`的构造器，这个配置对象有一个单一的属性`items`，它是一个会被实例化成`Ext.menu.MenuItem`的配置对象的数组。两个`MenuItem`的配置对象都会引用各自的处理程序来匹配文本项。

这个方法执行的最后两个函数是选择被右击的单元格，并在屏幕上正确的*X*和*Y*轴坐标显示上下文菜单。它通过调用网格面板行`SelectionModel`上的`select`❸方法，并传入选择的记录来做到这一点。最后，在右击事件发生的坐标位置显示上下文菜单。

在执行自己的代码之前，必须重新配置下网格来注册上下文菜单处理程序。把如下代码添加到网格配置对象里去：

```
listeners : {
     itemcontextmenu : doRowCtxMenu
}
```

现在已经有了开始使用UI新特性的所有东西了，接下来我们看看如何让其发挥作用。

8.5.4　使用创建和删除

此时此刻，已经开发完成了插入和删除处理程序，并且它们随时可供使用了。刚刚创建完成了上下文菜单处理程序，并重新配置了网格，让它可以在`rowcontextmenu`事件被触发的时候被调用。从创建和插入一条新记录开始探索，如图8-13所示。

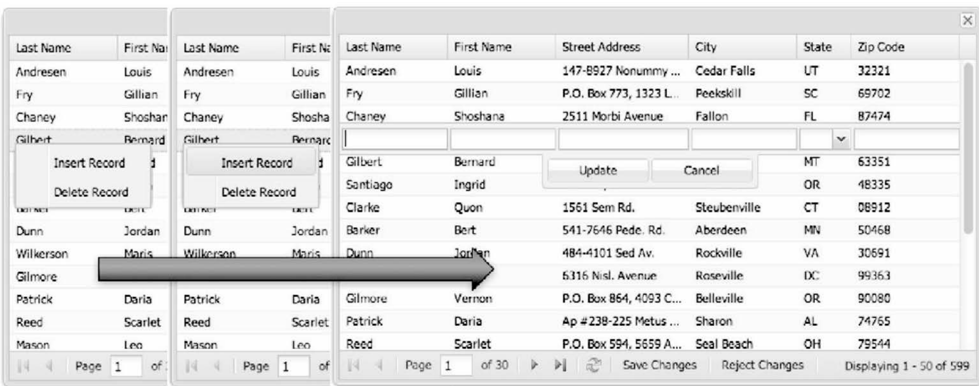


图8-13　使用新配置的插入记录菜单项来添加一条新记录

如图8-13所示，可以通过右击任意的单元格来显示上下文菜单，这会调用`doRowCtxMenu`事件处理程序。选择这就发生了，然后在鼠标所在的坐标位置显示一个定制的Ext JS菜单。点击Insert Record菜单项会强制调用注册了的处理程序`onInsertRecord`。`onInsertRecord`它会在被选择行的索引位置插入一条新的记录，并开始在第一列上进编辑。太棒了！

为了保存修改，需要更新新插入的数据记录，然后点击前面创建的保存修改按钮。图8-14展示了这一屏幕。哇！仅仅是创建记录就有这么多的东西。删除又是怎么样的呢？应该会简单点儿，对吧？

确实如此！在讨论删除记录的处理过程之前，让我们先看下UI是怎么工作的。

当右击记录的时候，一个`Ext.MessageBox`会弹出来询问是否确认删除操作，如图8-15所示。点击确认后，一个Ajax请求就会被发送到控制层去删除记录了。

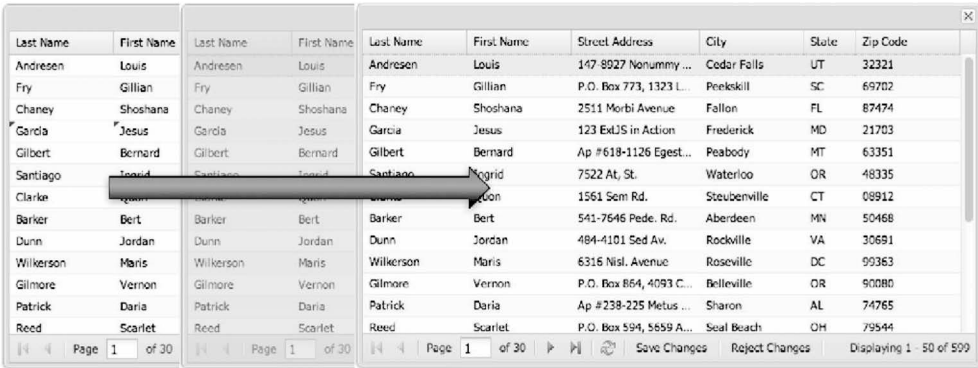


图8-14　保存新插入记录的UI变动图

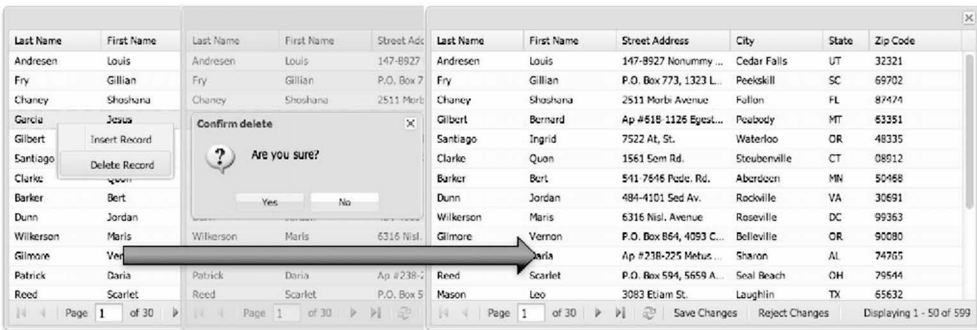


图8-15　删除一条记录的UI工作流

之所以如此是因为onDelete处理程序调用了MessageBox.confirm，并会调用onDelete方法。onDelete函数如下所示：

```
var onDelete = function() {
    var selected = grid.selModel.getSelection();
    Ext.MessageBox.confirm(
        'Confirm delete',
        'Are you sure?',
        function(btn) {
            if (btn == 'yes') {
                grid.store.remove(selected);
                grid.store.sync();
            }
        }
    );
};
```

这里使用了网格上的Employee Store（雇员存储）去删除一个选择了的记录，然后进行同步操作。

你现在已经为自己的第一个可编辑网格面板实现了所有的CRUD操作。在此过程中，你深入地了解了存储和记录，以及如何监测它们的修改并保存修改。一路上，你还有机会看到了Ext JS确认框发生的真实案例。

8.6 小结

本章介绍了不少关于网格面板以及在网格面板上使用数据存储的知识。当你构建第一个网格面板的时候，开始了解如何使用数据存储读取数据，如何做数据分页，甚至还实践了一把缓冲滚动分页。你还在网格面板上添加了网格交互操作，比如鼠标双击和右击会被捕捉到并使UI做出反馈。在此过程中，你还快速了解了菜单，并知道了及时在父组件被销毁之后清空菜单项的重要性。

最后，你首次接触到了`Grid`类，知道了如何使用RowEditing插件在系统运行时编辑数据。你还掌握了行的`SelectionModel`以及它的一部分方法，比如`getSelection`；也学会了使用键盘和鼠标手势在网格面板上导航，并更快地编辑数据。

现在，我们已经掌握了显示数据并与之交互的基本知识，下一章带你深入了解如何用图形显示数据。

第 9 章 深入探究树形面板

本章内容

- ❑ 剖析树形面板部件
- ❑ 渲染内存中的数据
- ❑ 使用远程加载数据的树形面板
- ❑ 创建自定义的上下文菜单
- ❑ 编辑节点数据

本章介绍Ext JS中树形面板的相关知识。树形面板用来显示层次化数据，与典型的文件系统非常相像。你将了解如何建立该部件的静态和动态实现。在熟悉了该组件后，你将通过使用一个数据存储实例和可动态更新的上下文菜单进行CRUD操作。

9.1 树形面板理论

要理解树，首先需要理解*层次*。层次是一种关系的组织形式，在该形式中，其中一项可能居于另外一项上层或下层，也可能两项居于同一层。在简单的实现里，层次是一对一的关系，但也可以是更复杂的多对多的有序集合。层次存在于社会、公司、军队、社交网络、教学和心理学中，当然也包含在计算机科学中。

9.1.1 树形面板关键词

你还需要了解这些与层次和树息息相关的特殊词汇。特定的概念是通过不同的名称来呈现的，因此我们要在深入主题之前预先定义这些专业术语。表9-1给出了Sencha中的术语。

表9-1 树术语

名 称	意 义
树形面板（Tree Panel）	一个拥有`Ext.panel.Panel`超强威力的容器，用于将层次化数据渲染成树形格式
树视图（Tree View）	一个负责渲染和操作树的DOM的组件
树（Tree）	代表项目（节点）的层次
节点（Node）	层次中一个项目。在Ext JS 4中，一个节点是通过`Ext.data.NodeInterface`来配置的

（续）

名　称	意　义
父节点（Parent）	作为参照物的节点所属的节点。根（Root）是最高级别的父节点
孩子节点（Child）	另一个节点的直接后代。所有节点都是根节点的孩子节点
根（Root）	包含了第一层和其他各层的所有节点（孩子节点）。根节点只有一个
叶子（Leaf）	没有孩子节点的节点
分支（Branch）	直接共享同一父节点的节点集合
深度（Depth）	从该节点的分支到根节点的距离。根节点的直接孩子节点具有一层的深度，而它们的直接孩子节点具有两层的深度，依此类推

在UI中，树这个词用来描述一个显示层次化数据的部件或控件，这些数据通常开始于同一个中心点，也就是根节点。就像植物里面的树一样，UI里面的树也有分支和叶子。和植物中的树不同的是，计算机领域的树只有一个根节点。

在计算机世界里，这种模式是无处不在的，它就存在于我们的眼皮子底下，只是我们不曾注意而已。你是否曾经浏览过计算机硬盘？（文件系统的）目录结构就是树形结构。它有一个根节点（Windows中的盘符）、分支（目录）和叶子（文件）。树同样也被应用于应用程序的用户界面上，并有其他一些为人所熟知的名字。

在其他用户界面（UI）库中，对这种类型的部件的其他命名还包括树视图、树UI或树，而在Ext JS中它称为树形面板。之所以称它为树形面板，这是因为它是`Panel`（面板）类的直接后代。很像网格面板，它不被用来包含任何子组件，除非这些子组件是专门为它设计的。`TreePanel`类之所以从`Panel`扩展而来，原因很简单：方便。这允许你使用`Panel`的所有UI精华，包括顶部和底部工具栏以及页脚按钮栏。

树形面板和网格面板共享原型继承。在Ext JS 4中引入的令人期待的更新不仅意味着树形面板现在更易于扩展，还意味着树形面板可以发挥数据包（data package）的全部潜力：存储（store）、读取器（reader）、代理（proxy）和写入器（writer）。特殊设计的类`TreeStore`可以为你管理层次化关系，简化CRUD操作。

9.1.2　深入根节点

为了让树形面板可以正常工作，首先需要一些数据。通过`TreeStore`实例加载数据，这会在内部解析关系并对`Model`实例应用所有与树相关的方法，比如`expand`（展开）/`collapse`（折叠）、`findChild`和`getPath`，以及维护树状态的额外项，比如`checked`、`allowDrop`和`parentId`。这些魔法会自动发生，通过`Ext.data.NodeInterface`类来实现。

数据加载完成后，树形面板将一个面板渲染为树的容器，并将其他的工作交与视图来做。`Ext.tree.View`负责每个节点繁重的UI输出工作。

下面才是棘手的部分。正如第1章中提及的那样，树形面板和网格面板共用相同的核心逻辑。乍听起来可能会觉得不可思议，但事实上这相当合理。它们均存在于`Ext.panel.Table`和

Ext.view.Table的上层。没错，树被封装在表格中，无论是在逻辑上还是在渲染上。

一棵完整的树代表一个表格，每个节点就像一行中的一列。默认情况下，一个节点代表一个文本和一个图标，但是可以被扩展以获得更高级的用法。属于一个Model实例的节点可以被赋予额外的任意数据。这些额外的信息可以用来在核心树的左侧或者右侧添加新列，实现一个经常需要的TreeGrid功能。整个处理过程如图9-1所示。

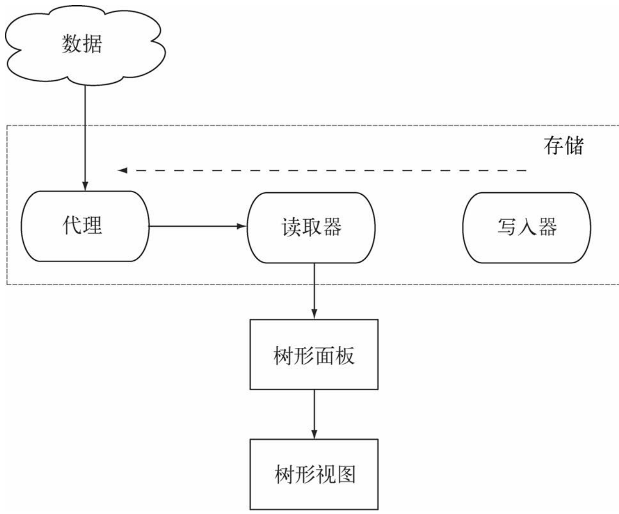


图9-1 树形面板的数据流和渲染周期

上升到一个特定层次，一个渲染出的树形面板将会共享GridPanel的CSS属性，比如x-grid-table、x-grid-row和x-grid-cell等。但是树的其他附加特性需要在专用的样式中指明，所有这些都由Ext.tree.View类来设置，该类从Ext.view.Table扩展而来。所以如果想为树节点添加一个自定义的界面外观，需要从这里入手。

现在你已经在一个较高的层面对树形面板及其工作机制有了了解，可以开始着手构建将从内存中加载数据的树形面板了。

9.2 “种下”你的第一棵树

实现一个和网格面板有“亲缘”关系的树形面板很简单。这里，将开始着手构建一个将从内存中加载数据的树形面板，这会让你对不久前所学的知识有更多的了解。下面的代码清单展示了如何构建静态的树形面板。

代码清单9-1 构建静态的树形面板

```
var store = Ext.create('Ext.data.TreeStore', {
    root : {
        text     : 'Root Node',
        expanded : true,
```

❶ 设置根节点

❷ 初始化时设置扩展属性为true

```
            children : [                                    ③ 指定孩子节点
                {
                    text : 'Child 1',
                    leaf : true
                },
                {
                    text : 'Child 2',
                    leaf : true                             ④ 指定叶子节点
                },
                {
                    text     : 'Child 3',
                    children : [
                        {
                            text     : 'Grand Child 1',
                            children : [
                                {
                                    text : 'Grand... you get the point',
                                    leaf : true
                                }
                            ]
                        }
                    ]
                }
            ]
        }
    });
    Ext.create('Ext.window.Window',{
        title       : 'Our first tree',
        layout      : 'fit',
        autoShow    : true,
        height      : 180,
        width       : 220,                                  ⑤ 配置树形面板
        items       : {
            xtype       : 'treepanel',
            border      : false,
            store       : store,
            rootVisible : true
        }
    });
```

代码清单9-1中的绝大多数代码都是支撑该树形面板的数据。浏览下根节点❶，会看到Root Node对象有一个text属性。这个属性非常重要，因为text属性正是Ext.view.Tree用来显示节点标签（label）的。在书写服务器端代码来支持这个部件的时候，请一定注意该属性。如果没有设置它，节点可能会显示在树形面板上，但不会有标签。

还会看到了一个expanded属性❷，它被设置为true。这项设置会保证在渲染时该节点被立即展开，并显示内容。在这里设置它，就可以在树形面板渲染后立刻看到根节点的孩子节点。该参数是可选的；如果不设置该参数，节点会被默认渲染为折叠状态。

Root Node对象❸被赋予children属性，这是一个对象数组。请注意Ext.data.NodeInterface会关注children属性，并在子数据（child data）有效时及时解析它们。当一个

节点有children数组时，数组中的所有对象会被转换为记录（Model）并填充到父节点的childNodes数组中。容器体系中也有类似的范式，其中子内容会被放在容器项目的MixedCollection中。

如果浏览了Root Node对象的children，就会发现第一个和第二个孩子节点并没有children属性，但是有一个被设置为true的leaf属性❹。把一个节点的leaf属性设置为true后，该节点将不包含任何子节点，这使它成为一个叶子节点而非分支。在这个例子中，Child1和Child2都是叶子节点，而Child3则是一个分支，因为它并没有一个被设置为true的leaf属性。

节点'Child3'包含一个子节点。该节点只有一个子节点，而这个子节点也只有一个子节点。最后一个子节点是一个叶子节点，因为它的leaf属性被设置为true。

在配置了支持数据后，接着使用XType❺配置对象来配置树形面板。在这里会看到这个部件的配置是多么简单自然。除了用来配置根节点的root外，其他所有属性都应该很容易理解。在这个例子中，根节点JSON最上层的对象会被看作树形面板的根。

可以通过为节点的配置对象添加icon或者iconCls属性来修改节点的图标，其中icon指定一张图像的直接路径，而iconCls是一个CSS类的名字，这个类用于指定图标样式。节点的iconCls属性和面板的iconCls配置对象作用很相似，我们更倾向于使用它们来更改图标。

最后，代码清单中的代码创建一个Ext.window.Window实例来显示树形面板。图9-2展示了渲染之后的树形面板。

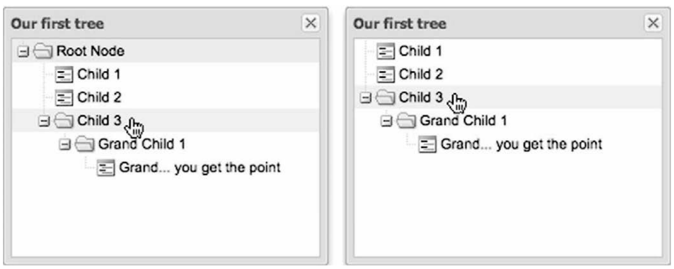


图9-2 第一个（展开的）树形面板：根节点可见（左），根节点隐藏（右）

在树形面板被渲染之后，可以看到在JSON中安排的那些节点就像设置的那样显示出来了。可以展开'Child3'节点和它的子节点来显示其他层级的节点。通过点击一个节点，你很容易使用选择模型。如果想隐藏根节点，可以将TreePanel配置对象中的rootVisible属性设置为false。在图9-2（右）中可以看到修改之后的结果。

现在有了它，一个静态的树形面板。很简单吧？现在，我们继续来创建一个远程树形面板吧。

9.3 培育动态树形面板

因为第一个树形面板是静态的，所以无需创建带有代理和读取器的完整的存储实例。远程加

9

载的树形面板改变了这些。在第7章，曾经使用一些数据在网格面板中显示人员身份信息，现在将使用相同的数据来开发一个树形面板。这些人都是My Company公司的雇员，不过分别属于不同的部门。

9.3.1　创建一个远程加载面板

接下来的代码清单显示了怎样配置一个树形面板，来使用服务器端组件列举每个部门的雇员。

代码清单9-2　构建动态的树形面板

```
var store = Ext.create('Ext.data.TreeStore', {
    autoSync: true,                                          ❶ 自动同步数据
    proxy   : {
        type : 'jsonp',
        url  : 'http://extjsinaction.com/treeData.php'       ❷ 远程数据URL
    },
    root    : {                                              ❸ 配置内联根节点
        text         : 'My Company',
        id           : 'mycompany',                          ❹ 设置根节点ID
        expanded     : true
    }
});

Ext.create('Ext.window.Window',{
    title       : 'Our first remote tree',
    layout      : 'fit',
    autoShow    : true,
    height      : 360,
    width       : 280,
    items       : {
        xtype       : 'treepanel',
        store       : store
    }
});
```

就像在代码清单9-2中看到的那样，要让存储知道，一旦有更改发生，它需要负责同步数据❶。然后配置了一个`Ext.data.proxy`❷，在本例中它是一个JsonP的代理，这个代理的`url`属性被设置为`treeData.php`，该页面被托管于http://extjsinaction.com。当需要配置自己的树形面板时，如果控制器和该文件拥有同样的域信息，可以使用自己选择的控制器替换掉该PHP文件，并把代理的类型改为Ajax。在开始编码控制器之前，我们先来了解下请求与响应的周期，在我们看完这个树形面板渲染出的版本后很快会讲到该点。

配置树形面板的下一步是配置内联的根节点❸。注意，节点额外的`id`属性❹十分重要。就像之后会看到的，该属性将用来从服务器请求子数据（child data）。还有一点要注意，将`expanded`设置成`true`。这样做可以保证一旦完成渲染，树形面板就会展开并加载子节点。

有趣的是，在`Ext.data.TreeStore`中根节点的配置并不是必须的。如果选择不去配置该项，请确保在最初的数据加载中包含它。这一特性在很多特定配置上都很有用。

最后，配置了一个更大的容器窗口（Ext.Window）来包含树形面板。将这个示例的窗口配置得更大不但可以增加树形面板的视图空间，还能消除对Ext.util.Format.ellipsis方法的潜在调用，因为命名过长。

在渲染完树形面板后，看到根节点（My Company）被立刻加载，并列出了My Company中所有的部门，就像图9-3（左）显示的那样。想要查看某个特定部门的员工，可以点击展开图标（+）或者双击标签，然后就会看到远程加载提示符显示在文件夹图标的位置，就像图9-3（中）显示的那样。一旦员工节点被成功加载，它们就会显示在部门节点下，就像图9-3（右）显示的那样。

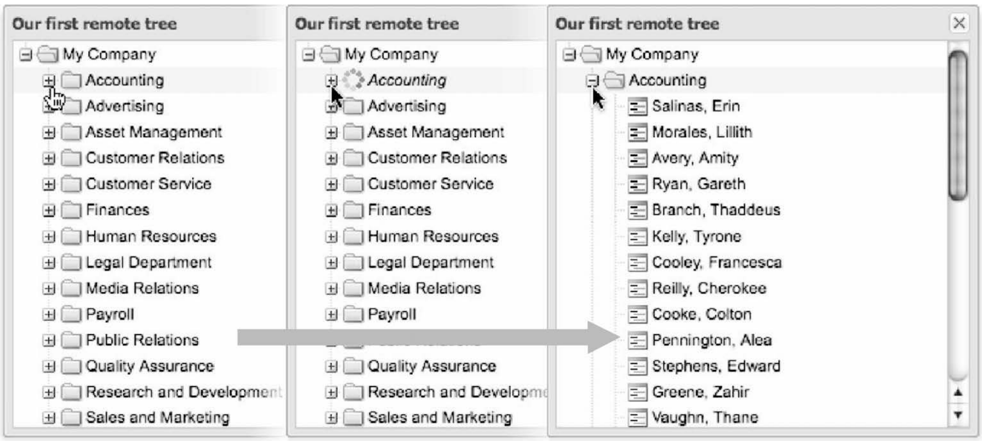


图9-3　远程树形面板显示了它远程加载数据的能力

我们快速地浏览了这个示例，现在稍微回顾一下，了解下请求是如何发出的。我们会讨论服务器端控制器是如何支持这个树形面板的。

9.3.2　为树（树形面板）“施肥”

为了使用树形面板来分析客户端/服务器端交互模型，让我们从根节点自动展开而触发的加载请求开始，如图9-4所示。记住将根节点的expanded属性设置为true，渲染的时候这个节点就会被展开，而如果子节点已经在内存中就会渲染子节点，否则就触发一个加载请求。

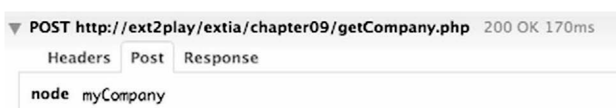


图9-4　初始节点请求的POST参数

就像看到的那样，发给getCompany.php控制器的第一个请求是由单个参数node（节点）构成的，并且参数值为myCompany。还记得你在哪儿设置了该值，将该值设置给了哪个属性吗？如果

你说是“根节点的id属性”，那就对了！当一个异步加载节点被首次展开时，存储将使用它的id属性来传递子数据给控制器。

控制器会接受这个参数并查询数据库，来获取与该id值相关的所有节点，然后返回一个对象列表，就像图9-5中演示的那样。在这个图中你可以看到一个用来定义多个部门的数组对象。每个对象都包含text属性和id属性。text属性会被赋给节点的label属性。注意部门数据中并不包含leaf属性和children属性。它们是叶子节点还是分支节点？



图9-5　初始请求getCompany.php控制器的结果

它们是分支节点。因为这两个属性都没有被定义，它们会被看作分支节点。这意味着当它们被初次展开时，TreeStore会调用一个Ajax.request请求，并传递该部门的ID属性作为node参数。控制器将接受该node参数并返回一个该部门所有雇员的列表。

使用刚才已经学到的知识，现在可以很安全地去预言，当展开Accounting部门节点时，一个指向getCompany.php的请求会被触发，它的唯一参数node被赋值为'Accounting'。让我们快速看下控制器请求的返回结果，它如图9-6所示。



图9-6　来自Accouting部门节点请求的结果

查看JSON结果，会看到返回的一个对象列表，每个对象都包含id、text和leaf属性。记住，因为leaf属性被设置了，所以这些节点都是不可再展开的叶子节点。

恭喜你！你已经成功地构建了能显示层次化数据的静态和动态树形面板。你对树形面板和提供树形面板数据的Web服务之间的客户端/服务器端交互模型也有了一个基本的了解。

如果致力于为这种类型的部件构建一个提供CRUD功能的UI，那么配置一个树形面板来加载数据只是其中的一小部分工作。接下来，我们看看怎样构建一个包含这些交互能力的树形面板。

9.4 在树形面板上实现 CRUD

为了配置包含CRUD功能的用户界面，需要添加更多的代码进去。毕竟，树形面板并没有原生地提供这些特性。这些就是你接下来要做的。

为了激活CRUD功能，需要修改树形面板，为它添加itemcontextmenu监听器，该监听器会调用一个方法来选择你右击的节点，并创建一个Ext.menu.Menu实例且将其显示在鼠标右击的坐标处。这一处理过程非常像在前一章中为网格面板添加上下文菜单处理程序的过程。

将创建三个菜单项：添加、编辑、删除。因为只能添加雇员到一个部门中，所以将根据点击的节点的类型来（根节点，分支节点或是叶子节点）动态地更改菜单项文本，启用或禁用该菜单中一些项。

每个处理程序都将调用存储中合适的CRUD API方法来为每个CRUD动作模拟控制器。因为存储自动完成了CRUD中的大多数工作，全部的数据创建和销毁处理过程和书中之前的例子很相似。

请准备好，这将是目前为止最复杂的树代码。首先，我们要创建上下文菜单处理程序和上下文菜单工厂方法。

9.4.1 显示上下文菜单

为了添加一个上下文菜单到树形面板中，必须为itemcontextmenu事件注册一个监听器。这一操作十分简单：在代码清单9-2中，添加一个listeners配置选项到Window创建代码里的items配置项下方，如下所示：

```
listeners   : {
   itemcontextmenu : onCtxMenu
}
```

添加这部分代码将会保证在itemcontextmenu（或者右键点击）事件被触发时onCtxMenu处理程序会被调用。

酷！现在树形面板已经设置好可以调用onCtxMenu处理程序了。在进行编码前，应该先构建一个工厂方法来生成一个Ext.menu.Menu实例，这将大大简化onCtxMenu。当使用工厂方法完成它的时候，将会更好地理解这句话的含义。下面的代码清单会让你了解如何构建上下文菜单的工厂方法。

代码清单9-3　配置一个上下文菜单的工厂方法

```
var onConfirmDelete = Ext.emptyFn;
var onDelete        = Ext.emptyFn;
var onEdit          = Ext.emptyFn;
var onAdd           = Ext.emptyFn;
var buildCtxMenu = function() {
     return Ext.create('Ext.menu.Menu',{
         items: [
             {
                 itemId  : 'add',
                 handler : onAdd
             },
             {
                 itemId  : 'edit',
                 handler : onEdit
             },
             {
                 itemId  : 'delete',
                 handler : onDelete
             }
         ]
    });
}
```

在代码清单9-3中首先设置了一些占位方法来指向Ext.emptyFn，这与实例化一个函数的新实例并无两样，只是看上去简单了许多。现在添加了这些占位方法，以后你想再来填充这些方法的时候，会清楚地知道在什么地方操作。

接下来，生成了buildCtxMenu工厂方法，该方法会返回一个Ext.menu.Menu实例，且会被接下来生成的onCtxMenu处理程序使用。以防之前并未见过或者听说过工厂方法，我们需要提示一下：其实，从一个更高的层次来看，它构造（因此名字里有个工厂）了一些东西并返回它构造的东西，这就是它的全部内容。

注意，没有一个菜单项包含text属性，但是它们均指定了itemId。这是因为onCtxMenu会动态地为每个菜单项设置text属性来给出用户提示：哪些事情是被允许的，哪些是不被允许的。itemId属性会被用来在菜单项的MixedCollection实例中定位具体的菜单项。

itemId这个配置属性和组件的id属性非常相似，只是它对组件的子容器来说是本地属性。这意味着与组件的id属性不同，itemId属性并不会被注册给ComponentMgr。也就是说，只有父组件有能力查看自己项目的MixedCollection，从而找到拥有某个特定itemId的子组件。另外，我们还可以在Ext.ComponentQuery中为某些id或itemId添加前缀#，例如#myComponent，然后执行这个查询很容易就能定位到相应的组件。

现在每个MenuItem都包含一个指向占位函数Ext.emptyFn的硬编码处理程序（作为占位方法），这样不用编写任何的真实处理程序，就可以看到自己的菜单显示在用户界面上。在开发了onCtxMenu处理程序并回顾了这个过程之后，将会继续为每个菜单项建立处理程序。onCtxMenu处理程序如代码清单9-4所示。

代码清单9-4　配置一个上下文菜单的工厂方法

```
var onCtxMenu = function(view, record, element, index, evtObj) {
       view.select(record);
       evtObj.stopEvent();
        if (! this.ctxMenu) {
            this.ctxMenu = buildCtxMenu();          ← ❶ 添加工厂方法
        }

        this.ctxMenu.treeNode = record;
        this.ctxMenu.treeView = view;

        var ctxMenu    = this.ctxMenu;
        var addItem    = ctxMenu.getComponent('add');
        var editItem   = ctxMenu.getComponent('edit');
        var deleteItem = ctxMenu.getComponent('delete');

        if (record.getId() =='mycompany') {          ← ❷ 配置节点的各种类型
            addItem.setText('Add Department');
            editItem.setText('Nope, not changing the name');
            deleteItem.setText('Can\'t delete a company, silly');

            addItem.enable();
            deleteItem.disable();
            editItem.disable();
        }
        else if (!record.isLeaf()) {
            addItem.setText('Add Employee');
            deleteItem.setText('Delete Department');
            editItem.setText('Edit Department');

            addItem.enable();
            editItem.enable();
            deleteItem.enable();
        }
        else {
            addItem.setText('Can\'t Add Employee');
            editItem.setText('Edit Employee');
            deleteItem.setText('Delete Employee');

            addItem.disable();
            editItem.enable();
            deleteItem.enable();
        }

        ctxMenu.showAt(evtObj.getXY()) ;
    }
```

在代码清单9-4中，创建了自己的onCtxMenu处理程序，它会使菜单变成动态的。这个处理程序要完成的第一个任务就是通过调用视图的select方法选择用户界面中的节点。这个select方法被继承自Ext.view.AbstractView的所有组件共享，例如Ext.gird.GirdPanel和Ext.picker.Time。选择一个节点是因为Ajax在被TreeStore的代理调用后，需要在树形面板

中进一步检索这个节点。

每当树形面板的itemcontextmenu事件被触发，它都将传递6个参数：

- 捕获该事件的视图；
- 代表一个Ext.data.NodeInterface记录的Model实例；
- 指向被右击节点的Ext.Element的引用；
- 该节点的数字索引；
- 生成的Ext.EventObject实例；
- 传递给Ext.util.Observable.addListener的options对象。

再返回到源代码，会发现并没有用到最后一个引用，因为不需要它。这里的引用太多了，不是吗？如果你觉得似曾相识，或许是因为这和网格面板中的itemcontextmenu事件非常相近。

接下来，通过显式调用evtObj.stopEvent终止了浏览器默认的上下文菜单。当需要用自己的上下文菜单替换浏览器默认的上下文菜单时，这样的处理方式会经常用到。

之后处理程序通过调用稍早之前创建的buildCtxMenu工厂方法来构建上下文菜单❶。它在本地保存了引用，就是this.ctxMenu，因此我们没有必要为随后出现的每个处理程序调用都创建一个菜单。

这时，想引用该视图和被选择的节点以便接下来能够使用它们。这样做避免了进一步的组件查询，以及危险的Ext.getCmp方法的调用。

之后创建了这个上下文菜单ctxMenu以及其每个菜单项的一个本地引用。这样做可以便于在以后需要管理菜单项时轻松读到它们。

在创建了本地引用之后，继续添加了一个if控制块❷，在这里检测节点的类型并据此修改菜单项。这是该处理程序的主体。下面是代码逻辑。

如果被右击的节点是根节点（record.getId() == 'myCompany'），配置菜单项以允许添加部门，但禁止删除和编辑公司节点的文本。还禁掉了这些菜单项（删除公司和编辑公司名）以使其无法被点击。毕竟，你不想任何人通过简单的鼠标点击毁掉整个公司的数据，不是吗？

继续，检测该节点是否是叶子节点（部门）。之后修改文本以允许添加雇员和删除整个部门。记住，如果需要的话，公司可以通过删除部门来做精简。还激活了所有菜单项。

如果被右击的节点是叶子节点，代码会执行else语句块。在这种情况下，add菜单项的文本会被修改并被禁掉，以防止添加一个雇员到另一个雇员，因为这种行为是不合理的。然后，修改并激活了“编辑”和“删除”菜单项文本。

最后，通过调用EventObject的getXY方法，获取到鼠标所在的坐标，并在此坐标显示上下文菜单。图9-7展示了每个被定制的菜单的外观。

如图9-7显示的那样，上下文菜单根据右击节点的类型显示不同内容，这演示了怎样通过一些修改使用相同的菜单完成相似的任务。如果不想显示或者想隐藏掉菜单项，而不是启用或禁用菜单项，可以将菜单按钮的enable调用切换为show，disable调用切换成hide。现在可以开始为上下文菜单添加处理程序了——可以从最简单的edit开始。



图9-7 显示公司（左）、部门（中）和雇员（右）节点的动态上下文菜单

9.4.2 添加编辑逻辑

你或许已经注意到了，点击一个菜单项后除了菜单消失以外并没有其他任何事情发生。这是因为虽然设置了上下文菜单，但是并没有真实的处理程序让其调用。现在将创建`edit`处理程序，这是到目前为止最容易编码的一个处理程序。

如果你已经熟悉了Ext JS 3或者更早的版本，甚至正在迁移一个可编辑的树到Ext JS 4版本，在这步就可能会注意到一个奇怪的特性：在更高的版本中缺乏编辑功能。

总是可以在一个表单或者窗口中绑定外部域，但是让我们再想一下。`Ext.tree.Panel`使用了`Ext.grid.Panel`的很多特性，而网格拥有那个很酷的`Ext.grid.plugin.CellEditing`插件（我们曾经在第8章中讨论过）。请尝试将它应用到树中。需要覆盖一个继承自`Ext.grid.plugin.Editing`类的方法，这个你已经很擅长了。让我们看下下面代码清单中的插件代码，可以将这个插件称为TreeCellEditing。

9

代码清单9-5 扩展CellEditing插件

```
Ext.define('TreeCellEditing', {
   alias: 'plugin.treecellediting',
   extend: 'Ext.grid.plugin.CellEditing',           ❶ 扩展Ext.grid.plugin.CellEditing插件

   init: function(tree) {
       var treecolumn = tree.headerCt.down('treecolumn');
       treecolumn.editor = treecolumn.editor
                     || {xtype: 'textfield'};       ❷ 分配编辑器

       this.callParent(arguments);
   },
    getEditingContext: function(record, columnHeader) {
        var me = this,
            grid = me.grid,
            store = grid.store,
```

```
            rowIdx,
            colIdx,
            view = grid.getView(),
            root = grid.getRootNode(),
            value;
        if (Ext.isNumber(record)) {
            rowIdx = record;
            //record = store.getAt(rowIdx);
            record = root.getChildAt(rowIdx);        ❸ 通过指定的索引找到被选择的节点
        } else {
            //rowIdx = store.indexOf(record);
            rowIdx = root.indexOf(record);           ❹ 通过记录引用找到被选择的节点
        }
        if (Ext.isNumber(columnHeader)) {
            colIdx = columnHeader;
            columnHeader = grid.headerCt.getHeaderAtIndex(colIdx);
        } else {
            colIdx = columnHeader.getIndex();
        }

        value = record.get(columnHeader.dataIndex);
        return {
            grid: grid,
            record: record,
            field: columnHeader.dataIndex,
            value: value,
            row: view.getNode(rowIdx),
            column: columnHeader,
            rowIdx: rowIdx,
            colIdx: colIdx
        };
    }
});
```

扩展的代码比它看上去更直接。现在做的事情我们将会在第13章进一步解释，所以请先耐心看下面的内容。在此，完成了两个目标。在称为`TreeCellEditing`❶的扩展类中，使用`AbstractPlugin`的`init`方法来检查该树是否有一个编辑器，并且如果测试失败的话为其赋予一个编辑器❷。只有想要自己定义树形面板的`columns`属性，并设置一个带有自定义编辑器的树列（tree column），才需要编辑器。因为树形面板自动创建了树列，只需要为其添加一个编辑器即可。现在，第一个目标完成了。

第二个目标看起来稍微困难一些。就像我们稍早之前提到的那样，需要修改的代码片段依赖于原型链的下一层，即`Ext.grid.plugin.Editing`。整个`getEditingContext`方法是通过两个简单的覆盖从代码中复制而来的。这个方法需要严格返回你选择的节点，无论传递给方法调用的第一个参数是谁，都可以通过它的索引❸或者记录引用❹来找到它。就在这两个覆盖方法的上面，可以看到我们曾经在网格中使用的源码。现在你已经准备好了自己的插件。

现在是时间让新插件开始工作了。Ext JS 4可以很容易添加这个新特性到树形面板，就像下面的代码清单中那样。

代码清单9-6 完成树的配置

```
var treeEditor = Ext.create('TreeCellEditing', {clicksToEdit: 2});    ❶ 实例化TreeCellEditing

Ext.create('Ext.window.Window',{
    title       : 'Our first remote tree',
    layout      : 'fit',
    autoShow    : true,
    height      : 360,
    width       : 280,
    items       : {
        xtype       : 'treepanel',
        store       : store,
        rootVisible : true,
        listeners: {
            itemcontextmenu: onCtxMenu          ❷ 为上下文菜单指定监听器
        },
        plugins: [
            treeEditor
        ]
    }
});
```

在代码清单9-6中，先初始化了TreeCellEditing插件。使用了`treeEditor`变量以便在树形面板中引用该实例。你本可以只使用`ptype`配置来做懒初始化，不过之后你会在编辑逻辑中获得直接引用插件带来的便利。为了完成显示，配置了`itemcontextmenu`监听器❷，就像之前提到的那样。

现在已经准备好构建编辑逻辑了。在此我们想提醒注意代码清单9-6中的`clicksToEdit`配置项❶。就像在可编辑网格中那样，可以使用已经注册过的`dblclick`事件进入编辑模式，而不用理会整个上下文菜单编辑的麻烦。但这是一个显示树形面板的内部交互的好机会。这就是为什么下面代码清单中你要在新的上下文菜单中初始化编辑器。

9

代码清单9-7 配置上下文菜单的编辑触发器

```
var onEdit = function(button, node) {
    var menu    = button.up(),                      ❶ 找到菜单
        node    = node || menu.treeNode,
        view    = menu.treeView,                    ❷ 访问引用
        tree    = view.ownerCt,
        selMdl  = view.getSelectionModel(),
        colHdr  = tree.headerCt.getHeaderAtIndex(0);
                                                    ❸ 获取列索引
    if (selMdl.getCurrentPosition) {
        pos = selMdl.getCurrentPosition();
        colHdr = tree.headerCt.getHeaderAtIndex(pos.column);   ❹ 显示编辑器
    }
    treeEditor.startEdit(node, colHdr);
};
```

在本章中你看到过更大块的代码，但是代码清单9-7中的看起来最为复杂。首先需要注册几个变量：通过检索被点击按钮的父容器❶而得到的`menu`；从之前定义（代码清单9-4）的菜单上

的属性获得的node❷和view；代表树形面板本身的tree；选择模型selMdl；列头部变量colHdr。选择模型会提供树形面板的列索引（请记住，树形面板也是表格，树视图是以列的形式渲染的，跟网格的行为很相似），因为在识别赋给树形面板的编辑器字段时列头部信息是必须的。回到node❷，请确认你已经注意到node也可以作为参数。稍后开始添加节点的时候你会用到它。

以防树形面板被配置为使用单元格集合模型（cellmodel），可以自动❸获取树列的位置，以确保正在编辑网格中正确的字段。最后，执行treeEditor插件实例的startEdit方法❹。注意到怎样重用treeEditor的引用了吗？

现在有了一个具备完整功能的树形面板，它可以为每个节点显示编辑器。当用户按下回车键时你认为会发生什么呢？你猜对了：记录会被更新，存储将会通过代理与服务器同步该更新。存储会独自承担这些工作。

刷新页面，右击一个节点，并请点击Edit按钮。在图9-8中我们右击选择该节点将Accounting部门的名字更改为了Legal。然后，我们点击Edit菜单按钮，它会在该节点的物理位置处渲染一个单行文本框为编辑器。接下来我们将名字从Accounting改为Legal并按下回车键。这将改变Node的值并触发存储的datachanged事件，自动调用sync方法。因为服务器接受了该值，新的值持久化在了用户界面上。请记住，当服务器返回{success: false}或者请求失败了，节点的文本值会被回置。



图9-8　使用树形编辑器编辑树形面板中一个节点的结果

这为树形面板提供了最简单的CRUD功能。在Web应用中编辑部件名是一件很常见的任务。很自然的，如何实现依赖于业务需求。使用新创建的TreeCellEditing插件可以简化应用流程，免于使用输入对话框，比如消息提示框。

接下来着手删除节点的事项。即使它需要额外的确认步骤，删除操作还是要比本章之前所做的要简单一些。

9.4.3　着手删除

为了设置树形面板的删除功能，要为Delete菜单按钮创建一个处理程序。很自然的，删除一

般需要向用户显示一个确认对话框，所以必须为用户确认进行编码。简单起见，将使用现成的Ext.Msg.confirm方法来显示这个对话框。这意味着需要为确认对话框构建一个回调函数。这个对话框回调函数将会触发存储的sync方法并最终删掉该节点。

现在你已经对自己需要做的事情有了一些了解，可以继续处理程序的编码了，就像下面代码清单中显示的那样。

代码清单9-8 向树形面板添加删除功能

```
var onConfirmDelete = function(answer, value, cfg, button) {
    if (answer != 'yes') return;                    ❶ 若点击了Yes，则返回

    var menu    = button.up(),
        node    = menu.treeNode;                    ❷ 移除、销毁节点
    node.remove(true);
};
var onDelete = function(button) {
    var callback = Ext.bind(onConfirmDelete, undefined, [button],
    ➥ true);                                       ❸ 创建回调函数

    Ext.Msg.confirm(
            'Approve deletion',
            'Are you sure you want to delete this node?',
            callback
    );
};
```

在代码清单9-8中，创建了两个方法来处理CRUD功能中的删除操作。第一个方法，onConfirmDelete，是晚些时候将创建的确认对话框的处理程序。如果对话框中的Yes按钮被点击❶，它将查找NodeInterface引用并删除它。设置了deletion参数为true❸，意味着它将会一并删除该节点❷。这一行为也会触发存储的datachanged事件，强制存储同步。

在这个虚构的迷你应用中，服务器将会获取到该节点的ID值，并在数据库或文件系统中执行删除操作，然后返回类似于{success: true}的信息，这代表着删除的确认。

创建的第二个方法（onDelete）是Delete按钮的处理程序。当这个方法被调用时，它会使用现成的Ext.Msg.confirm方法来显示一个确认对话框，并传递三个参数：标题、消息体和回调函数。

回调函数是使用工具函数Ext.bind（Ext.Function.bind的缩写）创建的。这个有用的函数创建了一个新的方法，该方法创建了一个在默认作用域内调用目标函数onConfirmDelete的闭包，并将button参数拼接到参数列表中。所以，Ext.bind将会收到四个参数：被调函数、作用域、参数数组，以及表示是拼接这些参数到参数列表（true）还是覆盖调用函数传递的参数（false）的布尔值。默认参数由Ext.Msg.confirm提供，它们是应答字符串（“yes”或者“no”），输入值（未定义，只有在使用Ext.Msg.prompt时有用）和confirm确认框的config对象。如果想使用anwser参数，那就全部保留它们。

现在应用展示给用户一条消息和两个选项。每个按钮都会触发回调函数，但是请记住，只有

当Yes按钮被点击时，才应执行节点删除操作。

请刷新用户界面并删除一个节点。图9-9演示了当我们刷新用户界面并删除Accounting部门节点时发生的事情。

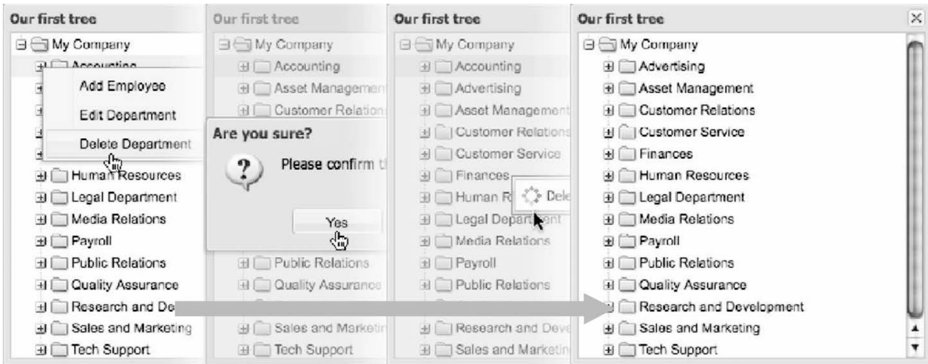


图9-9　通过确认框删除一个节点

在我们右击Accounting部门节点后，定制的上下文菜单出现了。然后，我们点击Delete按钮，它触发了`onDelete`处理程序。这立刻会显示确认对话框。我们点击Yes，这导致了`onConfirmDelete`方法被完整执行，查找存储在上下文菜单中的节点引用并删除（销毁）该节点本身。

在一个真实世界的应用中，删除一个分支节点通常会要求服务器递归地获取所有子节点的列表，并在删除分支节点前将这些子节点从数据库中删除。一个巧妙的做法是在数据库中设置一个触发器，当在某个容器节点上执行删除操作时通过触发器调用存储过程来删除与之相关的子节点。

因为我们通常需要一个确认对话框，所以从树形面板中删除节点还是需要花费一些努力的。添加一个节点，也同样困难，因为UI代码需要知道添加了哪种节点。是一个分支节点还是叶子节点呢？接下来，你将会看到怎么对这类行为进行编码，并让用户节点做出相应的反应。

9.4.4　为树形面板创建节点

为了创建一个节点接口，必须重用之前已经做过的一些工作，同时还要使用JavaScript的一些很酷的特性，比如闭包。因为树形编辑器需要绑定并显示在节点上面，将需要往树形面板中注入一个节点接口，并在这个新节点上触发一个编辑操作。一旦创建了该节点，存储就会将新数据同步到服务器。在创建节点时有机会编辑节点的名字，因此存储会在更新了文本（节点名字）后再次与服务器同步数据。这与在前一章中创建的编辑器网格中添加行的工作方式很相近。下面的代码清单包含了创建节点功能的代码。

代码清单9-9 向树形面板添加创建功能

```
var onAdd = function(button) {
    var menu    = button.up(),
        node    = menu.treeNode,
        view    = menu.treeView,
        delay   = view.expandDuration + 50,      <-- ❶ 确定动画长度
        newNode,
        doCreate;

    doCreate = function() {                       <-- ❷ 创建闭包
        newNode = node.appendChild({text: 'New employee', leaf: true});
        onEdit(button, newNode);
    };

    if (!node.isExpanded()) {
        node.expand(false,
           Ext.callback(doCreate, this, [], delay));   <-- ❸ 延迟节点的折叠
    }
    else {
        doCreate();
    }
};
```

在代码清单9-9中，你做了一些蛮有趣的工作，顺利实现了创建功能。下面是它的工作流程。

就像`onEdit`方法，需要从菜单中获取几个引用：`NodeInterface`和`View`。用后者来获取展开动画持续的时长❶。是的，用户界面动画和数据处理没有什么共同之处，但是这部分时间需要计入在内。当父节点需要折叠时，你需要先展开它，只有等到动画完成后，才能继续进行编辑处理。

间隔时长本身被隐藏在视图的配置中，可以增加额外的50 ms来让用户界面看起来更漂亮。遗憾的是，该框架并不会等待节点被展开，所以你需要在回调函数中添加足够的时延。

在规划`onAdd`处理程序的过程中，可以预见到两种可能的情形来获取展开或折叠的父节点。第一个情形很直接，可以在编辑过程完成后继续创建节点。但是如果该节点本来就是折叠状态的呢？在这种情况下必须展开它，并传递与创建和编辑`expand`方法的回调函数很相似的参数❸。因为重复这些代码是毫无意义的，创建了一个很好的闭包`doCreate`❷。闭包在这里非常有用；可以访问之前在父函数中赋过值的变量，甚至现在重用现成的代码也变得可能。

`doCreate`闭包封装了两个命令。第一个，将子节点拼接到被选中的节点的后面，并将其命名为New employee，将其`leaf`属性设置为`true`。记住，正在创建一个雇员，所以需要`leaf: true`。一旦完成了创建雇员，你会想去编辑它并赋予它一个合适的名字。这里重用了`onEdit`方法，传递相同的菜单按钮实例和新创建的节点给它，这样TreeCellEditing插件就知道在哪儿显示编辑组件。

哇，这个真是太有趣了！让我们刷新下用户界面来看看这些代码怎么工作，就像图9-10中显示的那样。在这儿能看到怎样使用TreeCellEditing插件来向树形面板中添加节点，就像操作系统一样。



图9-10　使用TreeCellEditing添加一个新节点到树形面板

当我们右击Accounting部门节点，动态的上下文菜单会像我们期望得那样出现。我们点击添加菜单按钮，它会触发`onAdd`处理程序。这会导致Accounting节点被展开。在所有子节点都被加载后，一个新的节点会被插入进来，`TreeStore`的`sync`方法会被触发，编辑操作会立刻在使用了TreeCellEditing插件的节点上被触发。我们键入一个雇员名字并按下回车键。这将导致`complete`事件被树形存储触发，进一步调用`sync`处理程序，这会执行另外一个XHR（`XMLHttpRequest`）。你现在还能实现代码来添加部门节点到树中，并向其中添加雇员。

做得太棒了！你知道了怎样构建树形面板，从服务器端为其获取数据，并为其添加CRUD流程。现在你能将这些部件添加到自己的应用中去了。

9.5　小结

本章，我们梳理了很多代码来讨论树形面板，以及怎样为节点建立很酷的CRUD交互。我们从`Ext.tree.Panel`聊起并讨论了一些支持类，比如`Ext.tree.View`、`Ext.data.TreeStore`和`Ext.data.NodeInterface`。你知道了`TreePanel`如何共享`GridPanel`的特性，从而比之前更加强大。你构建了一个静态的树形面板，在这里节点数据都是从内存中读取，你还看到了JSON是如何格式化的。

然后，你构建了一个动态的树形面板，并从一个远程数据源加载了数据，花了大量时间为其添加了CRUD操作。为了实现CRUD操作，你学会了动态修改、启用和禁用那个可以重用的上下文菜单。你知道了怎样使用自己的TreeCellEditing插件进行添加和编辑，这个插件还赋予树形面板内联编辑节点名字的能力。

至此，你已经了解了该框架提供的一些用于数据管理的部件，但这只是皮毛而已。下一章，你将会通过绘画和图表深入了解可视化数据的展现，学习它们的工作原理，并让它们在应用中更好地工作。

第 10 章

绘画和图表

10

本章内容

- ❑ 使用Ext.draw在浏览器中绘画
- ❑ 理解形状和表面
- ❑ 创建示例绘画
- ❑ 使用Ext图表和图表主题
- ❑ 配置图例

可视化数据展现被认为是最有效的用户体验机制。图表对于任何决策制定者来说都非常“养眼”，这就解释了为什么富有各种图表的仪表板经常是软件外观的前置特征。

Web应用也有同样的趋势。但是，无插件的图表支持才刚刚出现，在这个进程中Sencha扮演了趋势推动者的角色。

与其之前的版本相比，Ext JS 4对大量组件进行了升级。图表包不只被更新，而且被完全重写。它再也不需要Flash或者其他外部依赖来渲染令人惊艳的图表了。在Ext.draw包的支持下，Ext JS 4引入了新的图表类型，比如Scatter、Gauge和Radar。本章，你将学会实现有效实用的图表类型。

因为Ext JS图表的目标是不再依赖Flash来呈现“了不起”的图形，Sencha团队带来了可以使用线条和形状直接在浏览器中进行绘图的API。为了让你理解图表是怎样工作的，我们首先深入介绍新的Ext.draw包，告诉你图表是怎样被绘制出来的。请一定阅读本章中示例的相关文字，因为这是一个复杂的话题。

10.1 绘制形状

在浏览器世界中，很久以来可视化都是一个热门话题。支持令人激动的新特性的技术已经出现，之前的标准正在被废弃。企业级框架的一个重要使命就是支持像Microsoft Internet Explorer（IE）6那样老旧的浏览器。这意味着支持稳定但是令人乏味的JavaScript，或者我们应该称之为JScript，这是标准和可视化技术。

Ext JS绘图包支持可缩放矢量图形（SVG）和矢量标记语言（VML）。它们都对流行的浏览

器有很好的支持，并能推送矩阵图形到用户的屏幕。虽然VML比该框架支持更早的浏览器，但它缺乏SVG的灵活性。这就是为什么SVG被作为Ext.draw的默认绘图引擎。请注意，绝大多数IE浏览器都将需要使用VML。

如果你深入观察过Ext.draw，会发现它几乎是扮演了一个绘图引擎中间件的角色。它旨在提供一个统一的配置项，将命令推送到绘图引擎，绘制出相同的结果。注意，Ext.draw是如此地依赖于配置，有时甚至会打破和其他Ext JS组件的一致性。具体来讲，绘图时你必须引用某些特定的配置属性：

- `fill-opacity`
- `font-size`
- `stroke-opacity`
- `stroke-width`

注意，所有这些属性都包含一个连字符，而连字符只允许出现在JavaScript对象被引号包裹的属性中。

在使用这些属性之前，请随我们重温下浏览器内置矩阵绘图的基础。接下来，你将看到一些非常重要的概念。

10.2　绘图概念

绘图的前置条件是表面（surface）。我们故意不称之为画布（canvas），这是为了避免和HTML的画布标签混淆。一个表面就是一个接口，通常存在于一个`Ext.draw.Component`实例中。它提供了一个介于JavaScript和VML或SVG引擎的抽象层。

> **Ext JS 图表中的画布**
>
> Ext JS 4.1的图表只使用SVG和VML作为绘图引擎。Sencha Touch的图表提供了相似的接口，但是它们使用HTML5画布标签，来在移动设备上获得更好的性能。

表面，作为一个实例，在`Ext.draw.Component`实例中以属性的方式存在。这样，表面可以被用来绘制子画面（sprite）。子画面是用来组成形状的规则或不规则的路径。你很快会看到表面是怎样和子画面进行交互的。

10.3　表面子画面

没有表面绘图就无法进行，所以表面会一直被`Ext.draw.Component`使用，让我们检查下后者是如何处理艺术化的笔触的。

`Ext.draw.Component`直接继承自`Ext.Component`。所以，它共享了管理所有组件生命周期管道的能力。它最鲜明的特点，表面，可以适应你想要的各种形状。请记住，`Ext.draw.Component`的子类意味着各种形状。

只需要很少的几个配置项就可以定制一个Ext.draw组件。其中的每个配置项都会对绘图结果产生很大的影响。

- `autoSize`：将子画面定位到表面的左上角。虽然它不遵从子画面的*X*坐标和*Y*坐标，但是它们还是需要被设置。
- `viewBox`：设置为`true`来测量和定位项目以填充组件。覆盖大小和方位设置（*X*、*Y*、半径，等等）。
- `gradients`：一系列梯度，可以使用基于`gradientId`的子画面。
- `enginePriority`：指定想使用的首选绘图引擎，如果客户端浏览器支持的话。

一些选项会直接影响潜在子画面的行为。Ext.draw子画面支持的形状与绘图引擎原生支持的形状很类似：

- `circle`（圆形）
- `ellipse`（椭圆形）
- `rect`（rectangle，即矩形）
- `text`（文本）
- `path`（路径）
- `image`（图像）

当然，`path`是各行各业的千斤顶。你想绘制的所有形状都可以由`path`组成。稍后本章将会涉及`path`的语法基础。

子画面可以通过下面这些有用的属性做进一步定制。

- `width`：指定矩形的宽度
- `height`：指定矩形的高度
- `size`：指定正方形的边长
- `radius`：指定圆的半径
- `x`：指定在*X*轴上的左上角位置
- `y`：指定在*Y*轴上的左上角位置
- `cx`：指出圆形或椭圆形在*X*轴上的中心位置
- `cy`：指出圆形或椭圆形在*Y*轴上的中心位置
- `stroke`：指定边的颜色
- `stroke-width`：指定边的宽度
- `fill`：指定子画面主体的颜色
- `opacity`：指定子画面的不透明度
- `text`：包含要渲染的文本字符串
- `font`：为文本提供一个CSS样式的字体表述
- `path`：为绘制路径提供一个SVG语法的接口

针对子画面的特定形状，配置选项是可选的。例如，`size`只会被用于正方体，`radius`只会被用于圆形而不是其他形状。确切的匹配在框架的文档中有很好的描述。

10.3.1　绘制子画面

现在是时间来创建你的第一个艺术作品了。下面的代码单显示了怎样创建一个简单的圆形，并在维持纵横比的情况下对视图进行最大化。

代码清单10-1　绘制一个圆形

```
Ext.onReady(function() {
    var dc = Ext.create('Ext.draw.Component', {
         items   : [{
            type    : 'circle',                    ❶ 绘制圆形
            fill    : '#79BB3F',
            radius  : 100,
            x       : 200,
            y       : 200
        }]
    });

    Ext.create('Ext.window.Window', {             ❷ 创建父容器
        width       : 600,
        height      : 400,
        autoShow    : true,
        title       : 'Simple Circle',
        maximizable : true,
        layout      : 'fit',
        items       : dc,
        resizable   : {
            dynamic: true
        }
    });
});
```

代码清单10-1创建了最基本的图画。实质上，只在Ext.draw组件中创建了一个默认的表面，并将一个圆形的子画面置入其中❶。这个圆形的半径是100像素，它的*X*坐标和*Y*坐标均被设置为200像素。图像在`Ext.window.Window`中被渲染❷。如果你在浏览器中渲染这个示例，将看到一个如图10-1所示的图像。

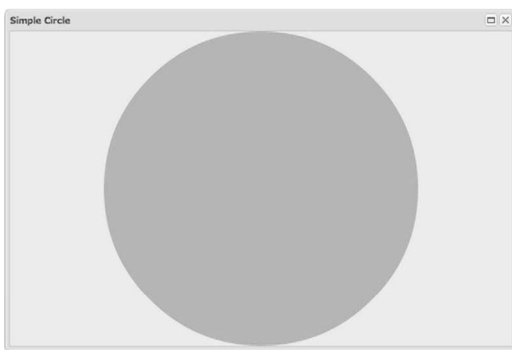


图10-1　你的第一个圆，渲染出来的效果

好样的！你已经使用Ext JS创建了第一个图画。在下面的部分，你会学习如何整合表面的设置和子画面的设置。这个示例将会参考代码清单10-1中的代码，并进一步扩展来证明一些至关重要的概念。

10.3.2　管理位置和大小

你已经成功创建了第一个图画，一个艳丽的绿色圆形。你训练有素的眼睛会立刻发现该圆形的半径超过了配置的100像素。原因隐藏在配置中了。它位于默认的`viewBox`配置项中，被设置为了`true`。这意味着这个圆形会被最大化到表面的大小，而忽视了`radius`的设置。

还有更多和配置项的交互。再次注意下`x`和`y`设置。`viewBox`设置并不会关心这些，只要它们被设置了。这听起来很古怪，所以我们会在一些示例中使用它们。

首先，分别将`x`和`y`设置为`1`：

```
x : 1,
y : 1
```

这个圆形还是被渲染为了同样的风格，虽然`x`和`y`都已经被更改了。这证明了`viewBox：true`覆盖了坐标。

现在我们尝试下不添加`x`和`y`的配置：

```
{
    type   : 'circle',
    fill   : '#79BB3F',
    radius : 100
}
```

讨论下这个古怪的行为。它令其中的一个坐标变为未定义，`viewBox`属性不再生效（见图10-2）。但是问题还是存在，我们到底应该怎样来设置圆形的半径和位置呢？

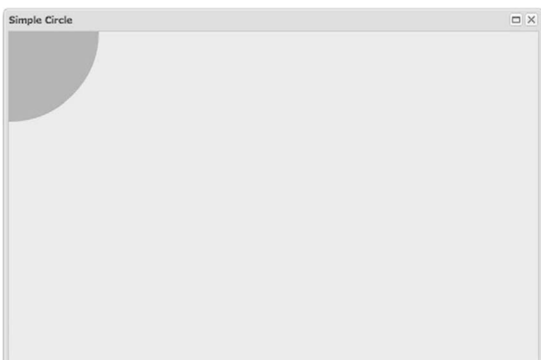


图10-2　不设置*X*轴和*Y*轴坐标，圆形的中心被设置为0,0，`radius`设置不再被忽略

有两种方式。第一种方式是将`viewBox`设置为`false`，就像下面代码清单中显示的那样。

代码清单10-2　禁用viewBox

```
var dc = Ext.create('Ext.draw.Component', {
    viewBox : false,                              ❶ 将viewBox设为false
    items   : [{
       type     : 'circle',
       fill     : '#79BB3F',
       radius   : 100,
       x        : 200,
       y        : 200
    }]
});
```

现在viewBox已经被禁掉了❶，让我们看看示例渲染以后的效果吧（参见图10-3）。

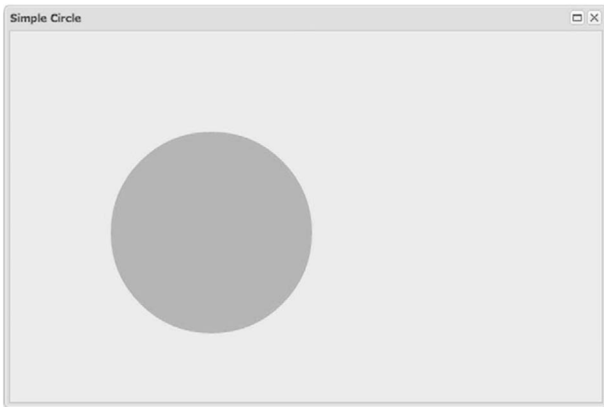


图10-3　现在圆形的位置和大小和期望的一致了

已经禁止了自动的大小和位置计算，取而代之的是固定的设置。从逻辑上来讲，圆形不应该关心x和y属性，因为圆形并没有左上角（其他角也一样）。Ext JS将会用坐标来匹配圆心。

第二种方法是设置*X*（cx）和*Y*（cy）属性，而不是x和y，就像下面代码清单中显示的那样。

代码清单10-3　用cx和xy有效定位圆形

```
items   : [{
    type     : 'circle',
    fill     : '#79BB3F',
    radius   : 100,
    cx       : 200,
    cy       : 200                  ❶ 将X、Y坐标居中设置
}]
```

用了更少的代码获得了同样的效果。设置中心*X*和*Y*❶和禁掉viewBox渲染出来的效果是一样的。

到现在为止，你只是把玩了一些简单的Ext.draw组件的行为控制配置：viewBox。在接下来的示例中，我们将展示autoSize怎样和你的子画面进行交互。

10.3.3　自适应大小的子画面

autoSize属性（查看代码清单10-4）会遵从子画面的大小设置，但是它会保证整个被渲染出来的画面会拥有可能的最小宽度和高度。很自然地，它排除了viewBox设置，但是它保留了x和y的设置需求。这意味着必须将x和y设置为整数，但将它们设置为多大的整数则无关紧要。

代码清单10-4　启用autoSize

```
var dc = Ext.create('Ext.draw.Component', {
      autoSize: true,                              ❶
      items    : [{
         type     : 'circle',
         fill     : '#79BB3F',
         radius   : 100,
         x        : 1,                             ❷
         y        : 1000                           ❸
     }]
});
```

❶ 将autoSize设为true

❷ 设置最小正整数

❸ 分配异常大整数

将autoSize设置为true❶，表面会自动将子画面定位到其内部，与其左上角对其(0, 0)，并确保占用最小的空间。为了演示当我们如此配置时，子画面是如何管理这个圆形的，将x设置为最小的正整数❷，将y设置为一个异常大的整数❸。你肯定注意到它是否被置于(1, 1000)。就如在图10-4中看到的那样，*X*和*Y*坐标都被忽略掉了，但是它们都需要被设置。否则，其渲染效果就和图10-2一样了。

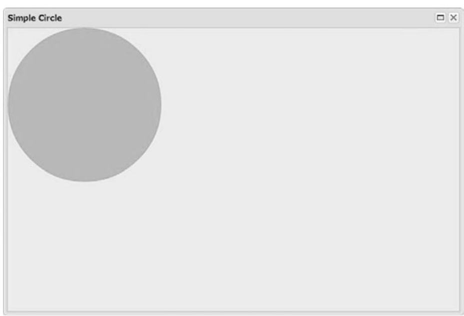


图10-4　该圆的绘制遵从了radius设置，但是被置于了表面的左上角

到现在为止，已经创建了一个简单的圆形，也已经看到了各种子画面和Ext.draw.Component（表面）配置的多种多样的组合。接下来要登场的是一些更令人激动的内容，怎样来动态地添加子画面和动画并绑定事件。

10.4　子画面交互

你可能想用Ext.draw来绘制一些更复杂的东西，而不仅仅是一个圆形。让我们往现有的组合里增加一些有趣的东西，来创建一个具有交互性的示例。

在下一个示例中（代码清单10-5），会重用之前示例中的圆形并将其置于一个固定的位置，禁掉viewBox。将动态地添加另一个具有淡入动画的圆形来获得一个更漂亮的登场。最后，会为新加的圆形添加一些事件监听器，使其当鼠标滑过时令其移动，在鼠标划出时回到初始位置。

代码清单10-5　动态添加一个形状

```
var dc = Ext.create('Ext.draw.Component', {        ❶ 实例化Ext.draw.Component
    viewBox : false,
    items   : [{
        type    : 'circle',
        fill    : '#79BB3F',
        radius  : 100,
        x       : 200,
        y       : 200
    }]
});

Ext.create('Ext.window.Window', {
    width        : 600,
    height       : 400,
    autoShow     : true,
    title        : 'Dynamically adding a new sprite to '
                   +'surface with a 2-sec delay',
    maximizable : true,
    layout       : 'fit',
    items        : [dc],
    resizable    : {
        dynamic: true
    },

    listeners: {
        show: function() {
            var sprite = dc.surface.add({          ❷ 访问表面以添加新的子画面
                type             : 'circle',
                fill             : '#846393',
                stroke           : '#a54222',      ❸ 控制线宽
                'stroke-width'   : 5,              ❹ 添加边框线宽
                opacity          : .8,             ❺ 控制线宽并填充
                radius           : 100,
                x                : 300,
                y                : 200
            });

            sprite.show();                         ❻ 在表面显示子画面
        }
    }
});
```

首先，创建了带有一个简单Ext.draw.Component❶子组件的Ext.window.Window，为绘图创建了一个表面（隐藏在dc.surface中）。该Ext.draw.Component被禁掉了viewBox和autoSize，因为你想自己控制子画面的大小和位置。第一个子画面被配置在items属性中，它是第一个圆形，就像之前示例中的那样。

在Ext.window.Window实例化刚完成后，开始添加另外一个子画面❷。将等待这个窗体来渲染和显示，因为只有在Ext.draw.Component渲染后表面才会被创建。你已经猜到了，在没有创建完的表面这一前提下不能添加子画面。

可以手动创建一个表面。Ext.draw.Component有一个createSurface()方法来完成创建表面的工作。现在，你知道了这个把戏，可以在将它们呈现给用户之前将其绘制出来。

回到我们的示例，新创建的圆形和之前的圆形拥有相同的大小，但是它被向左移动了100像素。它被赋予了不同的颜色，它的不透明度被设置为0。边线可以被配置为和填充体分离开并指定它们的颜色❸和宽度❹。不透明度的设置既会影响填充体颜色也会影响边线颜色❺。应该设置它们中一个的不透明度或者将它们设置为不同，可以一直获得CSS的威力，并能够使用RGBA来设置填充色和边线颜色（比如rgba(012, 123, 230, 0.4)）。

是时候来看一下子画面（参见图10-5）了。子画面的show()方法❻提供了可选的但是很方便的redraw参数。它告诉表面再次渲染子画面。现在你无法从redraw中获得更多便利，所以可以将它放在一边。



图10-5　第二个被动态添加的圆形

因为这个子画面的不透明度被初始设置为0，需要增大该数字来使这个圆形可见。为了让这个过程变得更酷，可以使用下面的代码来实现过渡动画，将它放在sprite.show()后面：

```
sprite.animate({
    duration: 1000,
    easing: 'easeOut',
    to: {
        opacity: .9
    }
});
```

不透明度的范围为0~1。设置为0.9将会使这个子画面看起来是完全可见的。该动画将会持续1 s（1000 ms），并且在接近最后时刻时减速。

现在你有了一个漂亮的登场。让我们来添加一些事件来让你的绘图可以响应鼠标指针移动。将下面的代码块添加到sprite.animate()代码块后面：

```
sprite.on('mouseover', Ext.bind(
    sprite.animate,
    sprite,
    [{
        duration: 500,
        easing: 'easeOut',
        to: {
            opacity: .6,
            translate: {
                x: -100
            }
        }
    }]
));

sprite.on('mouseout', Ext.bind(
    sprite.animate,
    sprite,
    [{
        duration: 300,
        easing: 'easeIn',
        to: {
            opacity: .9,
            translate: {
                x: 0
            }
        }
    }]
));
```

当鼠标指针移动到其上方向左划过时，第二个圆形会做出反应，就像图10-6所示。将鼠标移出该圆形，该子画面将会重置自己的位置。这两个事件的回调函数都移动了相应的子画面。

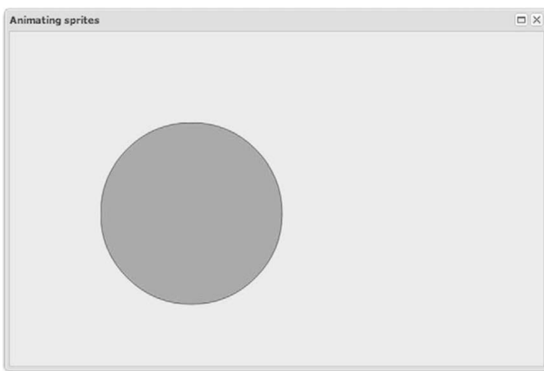


图10-6　鼠标悬浮时，第二个圆移动到第一个圆顶上

你刚刚已经学习了怎样动态地往表面中添加子画面，以及怎样实现动画和向它们注册事件。你或许已经发现这和预定义的图表类型共享模式。在下一节中，你将会看到图表是怎样绘制的，以及怎样使用路径子画面来创建一个自定义的形状。

10.5 掌控路径

SVG和VML是它们自己的语法引擎。到现在为止，在本章我们在不直接讨论语法的情况下成功地使用了这两个引擎。现在是时候接触创建自定义路径的基础知识了，这将是JavaScript世界外一次有趣的探险。

绘制路径本质上就是画一条从点*A*到点*Z*的线条，中间有*N*个点。它们的坐标是相对于*X*轴和*Y*轴的指定点。这条路径包围的区域也可以被定义为完全分开的样式。此外，两个点可以用一条直线相连，也可以被多种路径的曲线相连。这些行为通过下面这些命令来指示：

- M = 移动到
- L = 直线到
- H = 水平直线到
- V = 垂直直线到
- C = 曲线到
- S = 平滑曲线到
- Q = 二次贝齐尔曲线到
- T = 平滑二次贝齐尔曲线到
- A = 椭圆弧
- Z = 闭合路径

所有这些命令都可以用小写字母表达。大写字母指明是绝对定位点，而小写字母指明是相对定位点。

提示 关于矢量绘图更详尽的解释请参阅*HTML 5 in Action*（Manning，2013）。

示例是解释用法的最好方式。请准备一张纸和一支笔，现在画一个五角星（图10-7）。

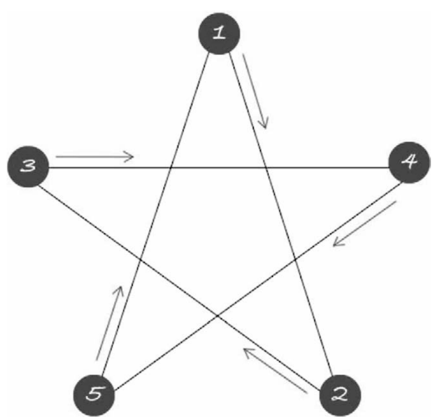


图10-7 通过一步一步的点连接手绘的星形

需要画多少条边？让我们借助这幅图来数一下。有起点，经过的4个点，回到起点，因此总共有5条边。

将这幅图转换为笛卡儿坐标系统。每个点都是由一对到原点（平面中心）的距离（用像素表达）的坐标组成的。让我们从0, –100开始：

```
M 0 -100 L 58 81 -95 -31 95 -31 -59 81 Z
```

第一眼看起来好像魔术，但其实它就是几何而已。用直白的语言表达如下：

(1) (M) Move to 0 –100
(2) (L) Line from 0, –100 to 58,81
(3) (Repeat L) Line from 58,81 to –95, –31
(4) (Repeat L) Line from –95, –31 to 95, –31
(5) (Repeat L) Line from 95, –31 to –59,81
(6) (Z) Finish the path and draw the line back to the starting position

(1)（M）移动到(0, –100)
(2)（L）从(0, –100)到(58,81)连接直线
(3)（重复L）从(58,81)到(–95, –31)连接直线
(4)（重复L）从(–95, –31)到(95, –31)连接直线
(5)（重复L）从(95, –31)到(–59,81)连接直线
(6)（Z）完成该路径并连接回起点

现在我们已经梳理过理论了，是时候将这段路径的代码插进我们的Ext JS 4应用中了，就像下面代码清单中显示的那样。

代码清单10-6　绘制一个星形

```
var dc = Ext.create('Ext.draw.Component', {
    items   : [{
        type    : 'path',
        fill    : '#ca433F',
        path    : 'M 0 -100 L 58 81 -95 -31 95 -31 -59 81 Z'
    }]
});

Ext.create('Ext.window.Window', {
    width       : 600,
    height      : 400,
    autoShow    : true,
    title       : 'Star (path)',
    maximizable : true,
    layout      : 'fit',
    items       : dc,
    resizable   : {
        dynamic : true
    }
});
```

就像你看到的那样，它的模式和其他子画面类型是一致的。在绘图组件中你添加了路径，但还需要额外的步骤才能真正完成路径的绘制。路径并不像圆形和正方形那样是预配置的，因此你需要用点指令来操作一下它。你要给path属性分配指令。输出效果请参见图10-8。

这个星型看起来就像手画的一样。此外，我们填充了它的内部，使得它看起来更致密。现在，你已经熟悉了创建和配置组件，所以唯一需要你添加进示例中的就是路径数据。它用一个字符串

的形式呈现，它就是这样传递给绘画引擎的。



图10-8　通过Ext.draw绘制的星形

现在你明白了在Ext JS 4中绘画的工作原理了。通过截至现在从本章学习到的知识，你现在明白了Ext.chart包是怎样在笛卡儿坐标系平面上绘制出图表了，甚至还可以创造出自己的图表类型。

在本章的剩下部分，你还会创建多种类型的图表。这将会是一次有趣的旅行，用Ext JS 4绘制动画、事件驱动的形状，以及线条和区域。

10.6　深入了解图表

在你绘制第一个图表之前，让我们先来仔细分析一个图表的主要组件。图表为存储中的数据提供了一个图形化的展现方式。再者，图表组件直接继承自`Ext.draw.Component`，这意味着你可以通过个性化的绘图为图表添彩。

就像图10-9展示的那样，图表的三个主要组件如下。

- **轴**（`Ext.chart.axis.*`）　定义数据维度。
- **序列**（`Ext.chart.series.*`）　可视化地展现一个数据项，如直线（点）、条、饼（扇片）等。
- **图例**（`Ext.chart.Legend`）　提供了一个出现在图表上的可选的变量列表，每个都代表着赋给数据点的子画面。

在基于笛卡儿平面的图表（折线图、条形图、散点图、柱状图、区域图）中，轴是指*X*轴和*Y*轴，分别代表着水平和垂直轴，每个轴都伴随着数字和分类标志，很容易从`Ext.data.Store`记录中提取出来。记录用数据来提供一系列的对图形化比较有意义的信息。

序列（series）代表了观察者的兴趣所在。它们是代表了特定值的绘画。在Ext JS图表的世界中，有8种有效的序列类型（见图10-10）：

10

- ❑ 折线图（line）
- ❑ 条形图（bar）
- ❑ 柱形图（column）
- ❑ 面积图（area）
- ❑ 散点图（scatter）
- ❑ 饼图（pie）
- ❑ 仪表图（gauge）
- ❑ 雷达图（radar）

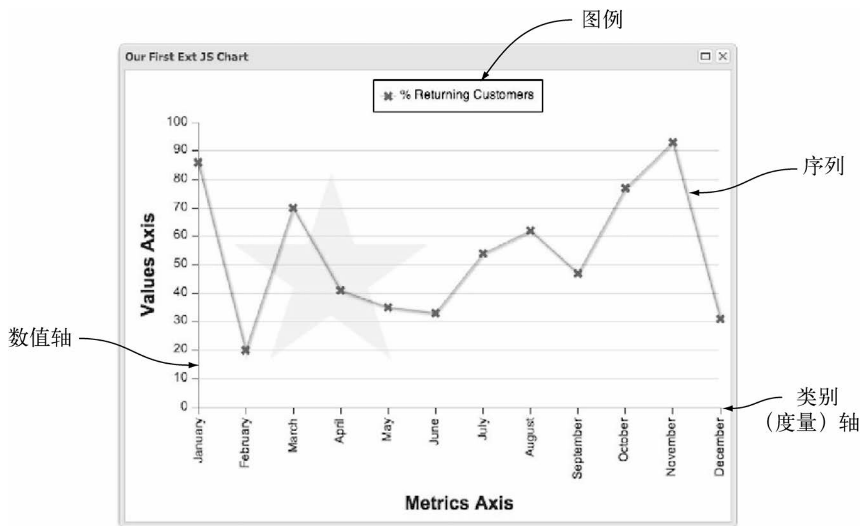


图10-9　第一个图表

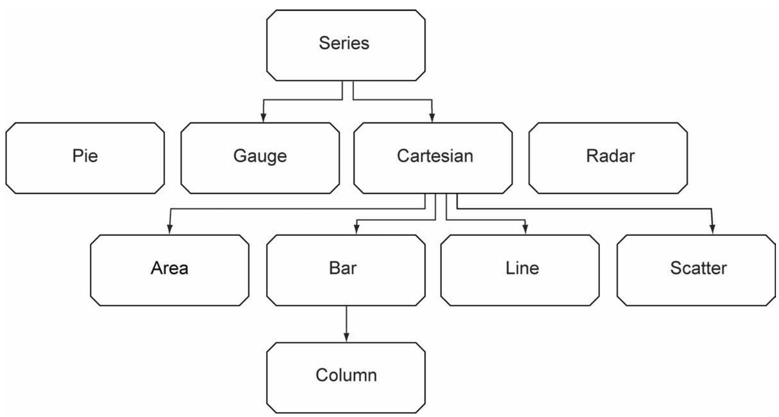


图10-10　序列继承模型。笛卡儿序列显示在这儿是因为它是Area、Bar、Line和Scatter类的超类，但是我们不会配置笛卡儿类本身的实例

让我们来审视下有效序列的列表和序列继承模型。如果这对于你来说太过复杂，别担心。你可以轻易地将它们归入两组：需要轴（饼型、刻度型和雷达型）的和不需要轴的（所有的笛卡儿类型）。

我们中的绝大多数人都会在某一刻面临一个抉择：到底哪种图表类型最适合用来呈现结果呢？这里有一些通用的规则来指导我们选用具体的序列。

- **线型**　显示大小跨度的数值改变趋势，在间隔较小时很有用。
- **条型**　用一个值和另一个进行比较，垂直排布。
- **柱型**　和条相同，但是呈水平排布。
- **面积型**　与线相似，特别是在跟踪两个或多个组的改变时很有用。
- **散点型**　用来决定两个不同值之间的关系，在可视化比重时很有用。
- **饼型**　不像笛卡儿类型，显示每片在整体中的比重。
- **仪表型**　在给定的最小和最大区间显示值。
- **雷达型**　并行比较由多个参数组成的各种选项。

因为图表是另一种表面，你很容易将定制的形状甚至图像添加到上面。但是请保证将它们放在面板上合适的位置，因为在图表中`viewBox`是`false`。你已经知道`viewBox`会实现自动的形状–位置计算。

现在我们已经介绍了基础知识，你已经知道怎样一步一步地创建如图10-9所示的图表。

10.7　实现笛卡儿图表

本章的早些时候，用了一个`Ext.draw.Component`实例访问了表面。在绘制图表的时候，将充分利运用这个直接继承自`Ext.draw.Component`的`Ext.chart.Chart`类，只是有一个重要的区别：`items`属性很少会被用到。稍后你会明白此处的原因。

10.7.1　配置轴

让我们再次从更基本的角度看下这个例子：手绘。当绘制草图的时候，你会先绘制什么呢？没错：轴。但是在开始绘轴之前，想一下到底期望什么种类的数据呢：最小最大范围、在每个轴上比较什么、如何度量空间。事不宜迟，让我们在下面的代码清单中引入轴。

代码清单10-7　列出轴

```
Ext.onReady(function () {
   var generateData = function (n, floor){      ← ❶ 添加随机数据生成器
       var data = [],
           i;

       floor = (!floor && floor !== 0)? 20 : floor;

       for (i = 0; i < (n || 12); i++) {
           data.push({
```

```
                name: Ext.Date.monthNames[i % 12],
                data1: Math.floor(Math.max((Math.random() * 100), floor)),
                data2: Math.floor(Math.max((Math.random() * 100), floor)),
                data3: Math.floor(Math.max((Math.random() * 100), floor)),
                data4: Math.floor(Math.max((Math.random() * 100), floor)),
                data5: Math.floor(Math.max((Math.random() * 100), floor)),
                data6: Math.floor(Math.max((Math.random() * 100), floor)),
                data7: Math.floor(Math.max((Math.random() * 100), floor)),
                data8: Math.floor(Math.max((Math.random() * 100), floor)),
                data9: Math.floor(Math.max((Math.random() * 100), floor))
            });
        }
        return data;
    };

    var store = Ext.create('Ext.data.JsonStore', {          ❷ 添加数据存储
        fields: ['name', 'data1', 'data2', 'data3', 'data4', 'data5'],
        data: generateData()
    }),

    chart = Ext.create('Ext.chart.Chart', {
        store: store,
        background: {
            fill: '#fff'
        },
        axes: [                                              ❸ 定义轴
            {
                type: 'Numeric',                             ❹ 设置轴类型
                position: 'left',                            ❺ 指定轴位置
                title: 'Values Axis'
            },
            {
                type: 'Category',                            ❻ 配置分类轴
                position: 'bottom',
                fields: 'name',
                title: 'Metrics Axis'
            }
        ]
    });

    Ext.create('Ext.window.Window', {
        width       : 600,
        height      : 470,
        autoShow    : true,
        title       : 'Our Very First Ext JS Chart',
        maximizable : true,
        layout      : 'fit',
        items       : [chart],
        resizable   : {
            dynamic: true
        }
    });
});
```

代码清单10-7有点儿长，因为它包含了很多我们在其他示例中已经使用过的东西。可复用的部分是随机数据生成器❶和数据存储（data store）❷。取元素而代之的是，必须设置轴（axis）❸。轴是一类很特殊的元素，它是`Ext.chart.axis.Axis`的实例。预配置的轴类型❹有：数字（数字）、类别（文本）、时间（日期对象）和刻度尺。

每个轴都需要被放置在笛卡儿平面的某个地方。在绝大多数的图表中，Ext JS的图表也不例外，观察一个简单的笛卡儿平面的四分之一就可以了。这就是为什么需要指明轴是作为该四分之一的哪条边界呢❺。可能的值为`left`、`right`、`top`和`bottom`。

最后，每个轴都应该被赋予一些数据。为了完成这个，需要从存储中引用一个字段，就像❻标注那样。请记住，在左侧轴上，并没有赋予一个字段。Ext JS会自动赋予一个从0到最大值的范围，相当于也指明了步长（step increment）。其中最大值是依据你使用序列中最大值计算出来的。如果没有指定序列，当然，在应用程序中也不应该出现这种情况——最大值会被设置为1。下一步在配置序列的时候会将其设置为一个固定值。

就像在图10-11中看到的那样，轴都像我们想象的那样被放置好了。数字轴（Values）像期望的那样被绘制出来并体现出来不指明序列所带来的缺陷。另一方面，Matrix轴缺少一些表现。February、September和November——它们中拥有最高值的名字——已经从该图中消失了。Ext JS会自动来分配空间以显示尽可能多的标签。

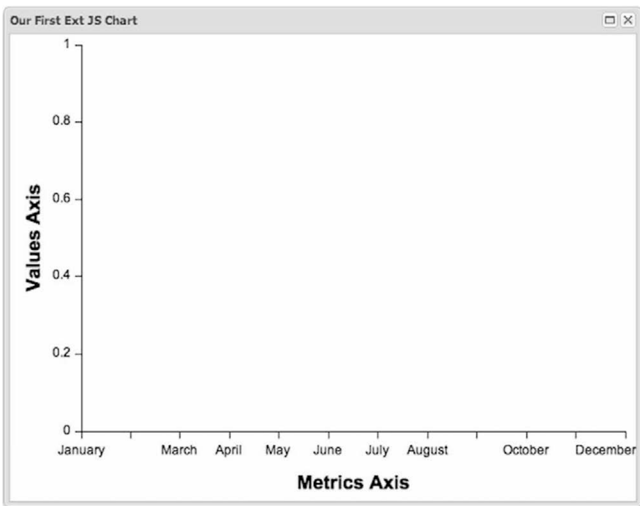


图10-11　绘制在表面上的轴

为了显示所有标签，必须旋转它们来垂直显示。像下面这样指明标签的配置：

```
{
    type: 'Category',
    position: 'bottom',
    fields: 'name',
    title: 'Metrics Axis',
```

```
        label: {
            rotate: {
                degrees: 270
            }
        }
    }
```

瞧，就和图10-9中演示的一样！除了旋转，标签可以被定制来显示指定的颜色和字体。你甚至可以使用一个渲染器自定义值的格式。

10.7.2　添加序列

除非为该图表添加序列，否则轴都是没用的。从随机生成的数据集中，会选取`data1`项并显示它是怎样随着月份（`name`属性）变化而变化的。接下来，在已经创建的`Chart`（代码清单10-7）的`axes`配置后添加了`series`的数组。

代码清单10-8　配置序列

```
    series: [
        {
            type: 'line',        ❶ 配置线型图
            axis: 'left',        ❷ 设置左侧轴
            xField: 'name',      ❸ 设置水平轴字段
            yField: 'data1'      ❹ 设置垂直轴字段
        }
    ]
```

首先，需要决定该图表的类型❶。序列的选择是线（line），会将它绑定到左侧轴❷。现在只需要配置存储字段中的名字来与*X*轴❸和*Y*轴❹相对应。在绝大多数情况下，后面这两项配置会与*X*轴和*Y*轴的配置相一致。图10-12显示了结果。

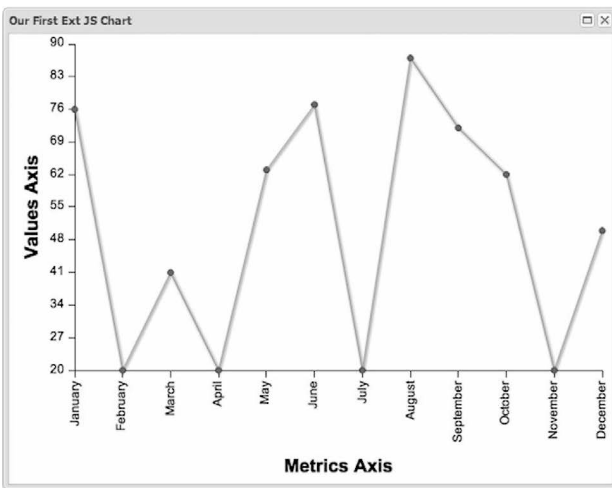


图10-12　Series就像配置的那样被绘制出来了

看起来很漂亮，不是吗？还请注意，月份标签在这里被垂直显示以获得额外空间。然而，图表还需要其他处理，比如在这种方式下很难比较5月份和10月份。如果使用网格线的话会容易得多，这是轴的一个功能，将向*Y*轴中添加它们。还需要用刻度线在*Y*轴中标识大小刻度。总体来说，需要10个大刻度，并且在每两个大刻度对之间还有5个小刻度。

再仔细看下图10-12，看看*Y*轴是从哪儿开始的。答案是：数字20。这个是自动（计算出来）的，因为这是存储记录中的最小值。如果真打算这么做的话，也可以强制设置该值。

10.7.3　改进可视化助手

伙计们，还没完。让我们来说下最令你感到挑战性的：确保*Y*轴显示保留一位小数的数字。现在，让我们来看下下面的代码清单。

代码清单10-9　在*Y*轴上改进可视化助手

```
{
    type          : 'Numeric',
    position      : 'left',
    fields        : ['data1'],
    title         : 'Values Axis',
    grid          : true,
    minimum       : 0,
    minorTickSteps: 5,
    majorTickSteps: 10,
    label         : {
        renderer: Ext.util.Format.numberRenderer('0,0.0')
    }
}
```

这个真的很直接，但是你没有使用可视化助手。让我们向图表中添加一些交互。首先，配置该序列来为每个值显示一个十字（而非点），就像下面代码清单中所做的那样。

代码清单10-10　定义标记配置

```
markerConfig: {
    type          : 'cross',
    size          : 4,
    radius        : 4,
    'stroke-width': 0
}
```

在代码清单10-8的这个序列的配置对象中，在`yField：'data1'`之后，创建了一个新的成员：`markerConfig`，它接受一个和`Ext.draw.Sprite`几乎完全相同的配置（作为参数）。唯一的区别是在`Ext.chart.Shape`单例中预定义的这个`type`属性。可选类型如下：

- ❑ `Circle`
- ❑ `Line`
- ❑ `Square`
- ❑ `Triangle`

- ❑ Diamond
- ❑ Cross
- ❑ Plus
- ❑ Arrow

每个十字都要比之前用过的点更大一些，这使得用户可以更容易地将鼠标放上去。让我们来为数字线序列添加下面的代码块来使其可以高亮显示（代码清单10-8）：

```
highlight: {
    size: 7,
    radius: 7
}
```

还可以设置highlight为true来显示默认的高亮分辨率，但是在最近定制的子画面中你指定了高亮的数值，而不是高亮分辨率。

如果要添加很好的提示来显示被选值的话，鼠标悬浮高亮将会变得更加有意义：

```
tips: {
    trackMouse: true,
    width     : 150,
    height    : 28,
    renderer  : function(record, item) {
        this.setTitle(record.get('name') + ': '
              + record.get('data1') + ' customers');
    }
}
```

现在在浏览器上跑一下这个图表，会看到如图10-13所示的内容。

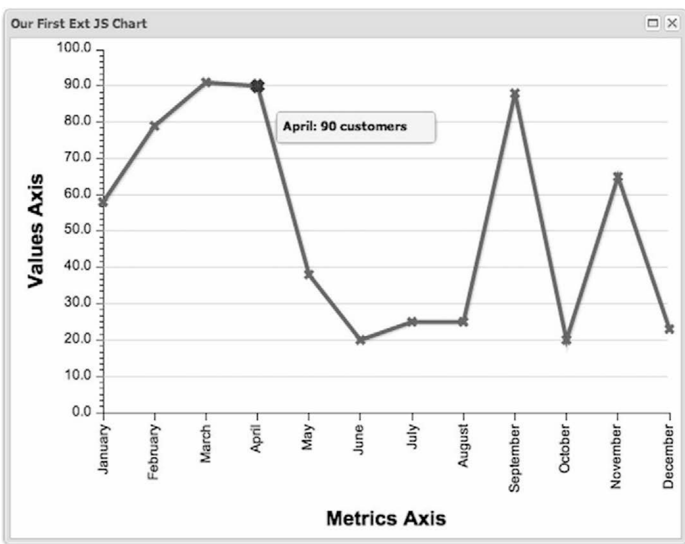


图10-13 对*Y*轴和序列的可视化改进

它看起来不错。记号标记和网格线都在这里，Value轴标签从0开始并显示了一位小数。十字已经取代了圆点，屏幕就像图10-13显示的那样，线是高亮的，强调了被选的十字，等等。最后，将鼠标指针指向一个十字将会显示它记录的值。

让我们将下面代码清单中的所有片段放在一起，来复习下我们用来显示图中图表的完整代码。

代码清单10-11 完整的线型图表配置

```
Ext.create('Ext.chart.Chart', {
    store: store,
    background: {
        fill: '#fff'
    },
    items   : [{
        type    : 'path',
        fill    : '#fff2cc',
        path    : 'M 200 100 L 258 281 105 169 295 169 141 281 Z'
    }],
    axes: [
        {
            type          : 'Numeric',
            position      : 'left',
            fields        : ['data1'],
            title         : 'Values Axis',
            grid          : true,
            minimum       : 0,
            minorTickSteps: 5,
            majorTickSteps: 10,
            label         : {
                renderer: Ext.util.Format.numberRenderer('0,0.0')
            },
        },
        {
            type: 'Category',
            position: 'bottom',
            fields: 'name',
            title: 'Metrics Axis',
            label: {
                rotate: {
                    degrees: 270
                }
            }
        }
    ],
    series: [
        {
            type: 'line',
            highlight: {
                size: 7,
                radius: 7
            },
            axis: 'left',
            xField: 'name',
```

```
            yField: 'data1',
            title: '% Returning Customers',
            markerConfig: {
                type: 'cross',
                size: 4,
                radius: 4,
                'stroke-width': 0
            },
            tips: {
                trackMouse: true,
                width     : 150,
                height    : 28,
                renderer  : function(record, item) {
                    this.setTitle(record.get('name') + ': ' +
                record.get('data1') + ' customers');
                }
            }
        }
    ]
});
```

在下一节，你将会用已经学习到的Ext.draw知识来定制一些形状并用它们来增强这个图表。

10.7.4　添加定制形状

你已经创建的图表是一个被子画面填充的表面。通过多种途径，已经访问过子画面的配置，不是通过通常的items属性，而是通过axis、series，甚至是markerConfig。这并不意味着items属性是不可访问的，事实正好相反。在下面的代码清单中，会使用它来添加一个定制的形状到图表中。

代码清单10-12　添加一个定制的子画面

```
items   : [{
    type    : 'path',
    fill    : '#fff2cc',
    path    : 'M 200 100 L 258 281 105 169 295 169 141 281 Z'
}]
```

items属性可以用和Ext.draw.Component相同的方式来使用。毕竟，图表继承自它。在这里使用了和代码清单10-6创建的相同的星形。比较这两个代码清单，你会注意到它们的不同点在于path属性。这是为什么呢？如果你想自己弄清楚请暂停一下。

当Ext.chart.Chart扩展Ext.draw.Component时，默认的viewBox配置为false。这意味着在*X*轴和*Y*轴平面没有自动定位。取而代之的是，每一个坐标都是绝对的，必须依此自己安排自己的子画面。图10-14显示了在图表中的星形。

让我们进一步定制形状，并用星形替换掉十字（代码清单10-13）。这次一个简单的配置就不够了。必须在Ext.chart.Shape单例中注册一个新的形状，这将是一个使用Ext JS新的类系统的好做法。

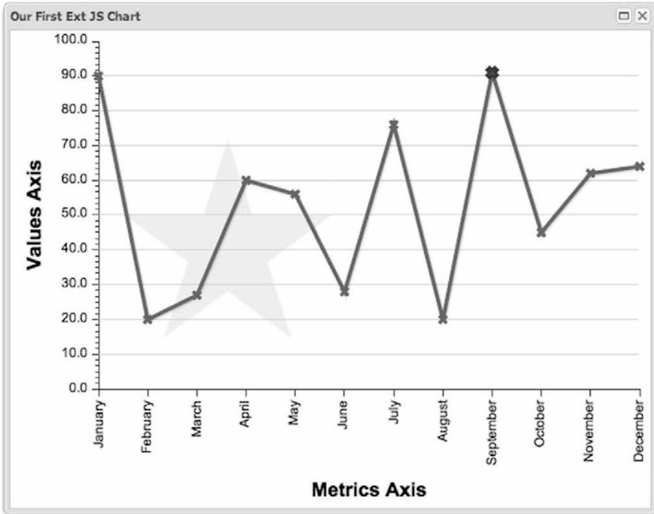


图10-14　在一个图表上渲染出的定制形状

代码清单10-13　定制标记安装

```
Ext.chart.Shape.self.override({
    star: function (surface, opts) {
        return surface.add(Ext.applyIf({
            type           : 'path',
            path           : 'M 0 -10 L 6 8 -9 -3 10 -3 -6 8 Z',
            'stroke-width' : 0,
        }, opts));
    }
});
```

为了访问一个单例的`override`属性——这是Ext JS新的类系统带来的糖果——必须首先访问它的`self`属性。在代码清单10-13中，将添加一个新的方法（`star`），它将代表一个新的标记配置类型。

路径需要再次调整。这次每个星形都应该放置到每个点的中心，因此也就是像素点的调整。最后，在每个星形绘制完成后没有中断边线。可以通过给`star`方法传递`opts`参数来访问所有的这些子画面。让我们将`star`插入到标记配置中：

```
markerConfig: {
  type: 'star'
}
```

不需要进一步的配置了，这就是全部内容。现在跑一下新的代码，应该会看到如图10-15所示的内容。

你经常想比较一下两个或多个不同数据集相对于一个普通值的关系，比如月份。我们接下来的讨论将会关注这一点。

10

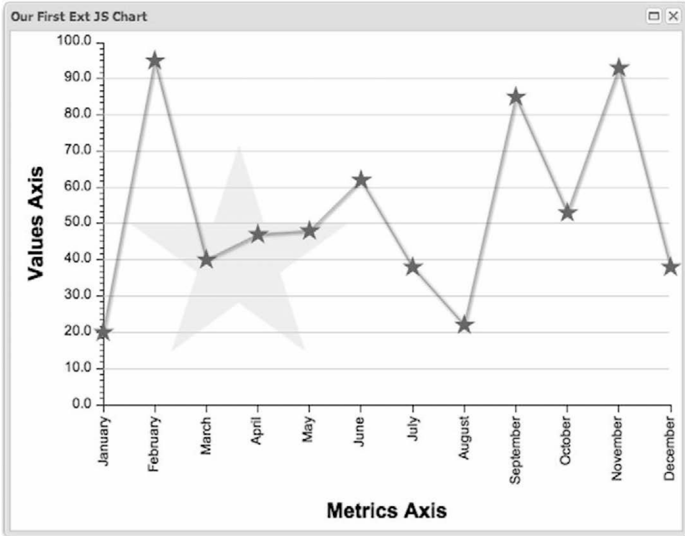


图10-15 星点

10.7.5 同一个图表中的多序列

Ext.chart.Chart中的序列配置是一个Ext.chart.series.Series实例的数组。换句话说，一个图表可以包含合理数量的序列。这在共享*X*和*Y*平面的笛卡儿类型图表中通常都是合理的。

重用之前已经完成的代码，添加一个新的序列配置（参见代码清单10-14）来代表客户的数量相较于回头客户的数量。

代码清单10-14 添加序列

```
,{
    type: 'line',
    highlight: {
        size: 7,
        radius: 7
    },
    axis: 'left',
    xField: 'name',
    yField: 'data2',
      markerConfig: {
        type: 'diamond'
    }
}
```


就这么简单，已经添加了一个新的配置对象来负责新的序列。你已经很熟悉这个配置了。主要的变化是yField数据资源❶。为了使点更容易被追踪，这个序列将会绘制菱形❷而不是值（参见图10-16）。

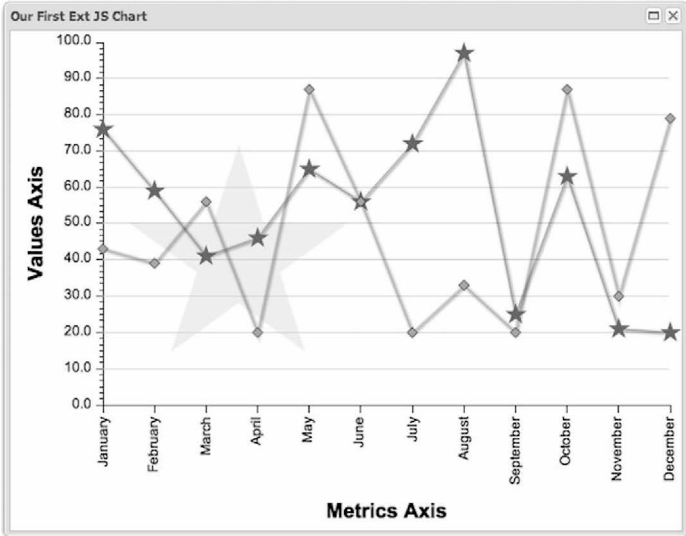


图10-16 多线型序列

看起来不错，不过有点儿乱。首先，哪条线代表哪组数据？调用图例来拯救它吧！

```
legend: {
    position: 'top'
}
```

开启图例并将其放置在顶部，需要为必需的图例分配一些垂直空间。图例会用一个标记样例来说明序列的颜色和名称。如果标题没有被使用，那么相应的字段名就会被显示，就像在Ext.data.Model中指明的那样。做如下的两处更改：

```
{
...
    title: '% Returning Customers',
...
},
{
...
    title: '% New Customers',
...
}
```

每一个标题都在各自的序列配置当中。这种更改在图形清晰度方面可能不会令人满意，也正因此会将第二个序列（% New Customer）置于一个面积图中：

```
type: 'area'
```

这次改变全部完成了吗？当然！请参见图10-17。

10

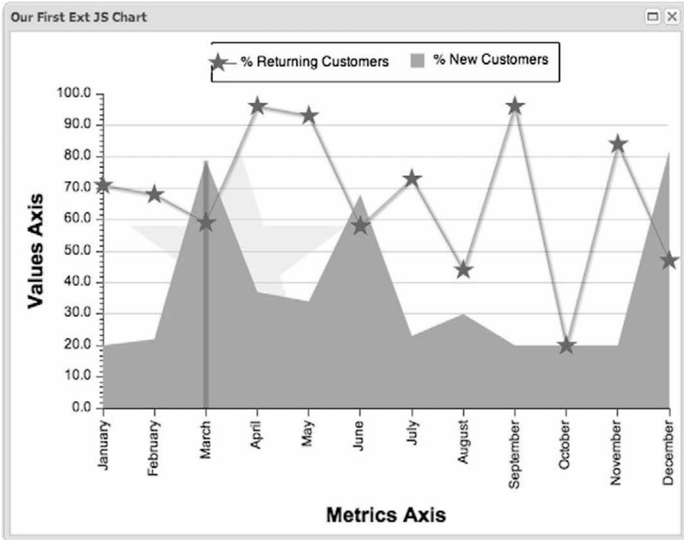


图10-17　一个组合图表中的图例

更具有可读性了，不是吗？图例很好地显示了每个序列的标题，通过使用新的序列类型让这两个数据集的差别更加明显了。面积图有着不同的高亮机制，它会显示一条细小的黑色线条来显示其在水平平面的位置。

所以，现在有了两个序列：一个蓝色的线型图和一块儿绿色的面积图。然而，我们从没提到过任何关于颜色的东西，这怎么得了？很快你会发现其中的小秘密。

10.8　定制主题

图表，作为数据的一种可视化表示，依赖于图形形状和颜色。轴和序列满足了形状的需要，但是怎样来控制颜色呢？答案就在Ext.chart.theme包中。

这个包中有两个主要的类：Ext.chart.theme.Base和Ext.chart.theme。Theme.Base是私有的，它包含了默认的主题配置。Theme类用来定义新的主题。

为了更改地解释该配置，先浏览下默认的设置。下面的代码清单显示了默认配置对象。这个代码清单实在是很长，所以请保持注意力。如你所见，配置图表需要做很多工作，主题又带来了新的工作量。

代码清单10-15　主题配置

```
{
    background: false,
    axis: {
        stroke: '#444',
        'stroke-width': 1
    },
```

```
axisLabelTop: {
    fill: '#444',
    font: '12px Arial, Helvetica, sans-serif',
    spacing: 2,
    padding: 5,
    renderer: function(v) { return v; }
},
axisLabelRight: {
    fill: '#444',
    font: '12px Arial, Helvetica, sans-serif',
    spacing: 2,
    padding: 5,
    renderer: function(v) { return v; }
},
axisLabelBottom: {
    fill: '#444',
    font: '12px Arial, Helvetica, sans-serif',
    spacing: 2,
    padding: 5,
    renderer: function(v) { return v; }
},
axisLabelLeft: {
    fill: '#444',
    font: '12px Arial, Helvetica, sans-serif',
    spacing: 2,
    padding: 5,
    renderer: function(v) { return v; }
},
axisTitleTop: {
    font: 'bold 18px Arial',
    fill: '#444'
},
axisTitleRight: {
    font: 'bold 18px Arial',
    fill: '#444',
    rotate: {
        x:0, y:0,
        degrees: 270
    }
},
axisTitleBottom: {
    font: 'bold 18px Arial',
    fill: '#444'
},
axisTitleLeft: {
    font: 'bold 18px Arial',
    fill: '#444',
    rotate: {
        x:0, y:0,
        degrees: 270
    }
},
series: {
    'stroke-width': 0
```

```
        },
        seriesLabel: {
            font: '12px Arial',
            fill: '#333'
        },
        marker: {
            stroke: '#555',
            radius: 3,
            size: 3
        },
        colors: [ "#94ae0a", "#115fa6","#a61120", "#ff8809", "#ffd13e",
         "#a61187", "#24ad9a", "#7c7474", "#a66111"],
        seriesThemes: [{
            fill: "#115fa6"
        }, {
            fill: "#94ae0a"
        }, {
            fill: "#a61120"
        },
        ...
        ],
        markerThemes: [{
            fill: "#115fa6",
            type: 'circle'
        }, {
            fill: "#94ae0a",
            type: 'cross'
        },
        ...
        ]
}
```

你已经知道了图表是子画面的一个分支，所以添置一个新的主题要求很多Ext.draw技巧。让我们复习下这些配置属性。

- background（背景）：始终居于底层的填充，拥有最低的z-index。
- axis（轴）：一条线，代表一个轴。
- axisLabelTop（顶部轴标题）、axisLabelRight（右侧部轴标题）、axisLabelBottom（底部轴标题）、axisLabelLeft（左侧轴标题），分别是每个轴位置的标签。
- axisLabelTop（顶部轴标题）、axisLabelRight（右侧部轴标题）、axisLabelBottom（底部轴标题）、axisLabelLeft（左侧轴标题），都是轴标题。
- series：默认轴配置。
- seriesLabel：序列中每个值的标签。
- marker：默认标记。
- colors：十六进制颜色，序列的数组索引和颜色的数组索引相同。
- seriesThemes：序列的绘图配置，同colors的数组配置规则相同。
- markerThemes：标记的样式，和colors的数组配置规则相同。

我们确信，理解Ext.draw.Sprite的配置将会在创建新的主题时更加轻松。让我们来为图表实现一个定制的主题，如代码清单10-16所示。

代码清单10-16 配置春天主题（Spring）

```
Ext.define('Ext.chart.theme.Spring', {          ❶ 定义春天主题类（Spring）
    extend: 'Ext.chart.theme.Base',
    constructor: function(config) {
        var axisColor = '#610519';

        config = Ext.apply({                      ❷ 扩展基本主题类（Base）
           axis: {
               fill: axisColor,
               stroke: axisColor
           },
           axisLabelLeft: {
               fill: axisColor
           },
           axisLabelBottom: {
               fill: axisColor
           },
           axisTitleLeft: {
               fill: axisColor
           },
           axisTitleBottom: {
               fill: axisColor
            },
           colors: ['#6695CC', '#65ed73'],

           seriesThemes: [{
                fill: "#CC7200",
                stroke: '#3FCC00'
            }, {
                fill: "#610519"
            }],
            markerThemes: [{
                stroke: '#f00'
            }]
        }, config);
        this.callParent([config]);
    }
});
```

将该主题称为Spring❶。一个新的类必须扩展Ext.chart.theme.Base❷，这是被强制的。之后，事情变得直接了。绝大多数工作（参见图10-18）都是用fill（填充）和stroke（边线）颜色实现的。

10

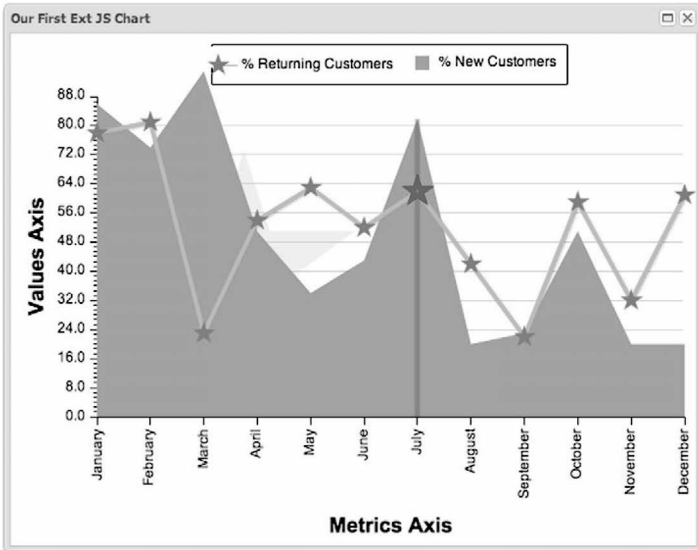


图10-18　运行的新主题

当我们准备讨论图表的时候，往往都已经讨论过笛卡儿类型。它们共享一个相似的配置，它们都包含轴。接下来，我们尝试实现一个无轴的图表。

10.9　饼图

当我们要观察一个值在总体中的比重时，经常会换用饼图。它们不会从轴中得到便利，但是会和数据集中的其他数值进行交互。在这个例子中，你将会重用相同的数据存储，但限制了数据的数量。在同一个序列中存在太多的数据会使图表非常凌乱。

饼图会有一个图例和在每个分片里被写入的标签。分片将会对鼠标悬浮做出反应并将所选分片移出边界，就像代码清单10-17中那样。

代码清单10-17　配置数据存储，渲染图表

```
var store = Ext.create('Ext.data.JsonStore', {
    fields: ['name', 'data1'],
    data: generateData(4,30)
});

Ext.create('Ext.chart.Chart', {
    animate       : true,
    store         : store,
    shadow        : true,
    insetPadding : 10,
    legend        : {
        position : 'bottom'
    },
```

❶ 生成小数据集

❷ 设置图表填充

```
    background    : {
        fill : '#fff'
    },
    series        : [
        {
            type         : 'pie',
            field        : 'data1',
            donut        : 40,
            showInLegend : true,
            tips         : {
                trackMouse : true,
                width      : 150,
                height     : 28,
                renderer   : function (record, item) {
                    this.setTitle(record.get('name')
                           + ': ' + record.get('data1'));
                }
            },
            highlight    : {
                segment : {
                    margin : 20
                }
            },
            label        : {
                field    : 'name',
                display  : 'rotate',
                contrast : true,
                font     : '18px Arial'
            }
        }
    ]
});
```

❸ 配置饼图类型

❹ 连接模型

❺ 在饼图中添加“孔”

❻ 图例中显示类别

使用在本章早些时候创建的数据生成器，创建了一个小型随机数据集❶。然后，创建了一个图表并保证为饼留够空间❷来完成鼠标悬浮效果。没有创建任何轴，这没问题，给定的序列类型是pie❸。*X*和*Y*都不存在，所以没有xField和yField，只需要模型的field名赋给field就可以了❹。“孔”是一个donut，赋一个以像素为单位的值来作为它的半径❺。最后，指示序列需要显示图例❻。

看起来需要花费一些功夫，但是一旦你学会了创建基本的图表，就可以利用这些知识创建任何其他类型的图表。饼图与笛卡儿类型的图表最大的不同在于与轴的关系。智能配置使它们更加简单。让我们来看看刚刚做了什么，参见图10-19。

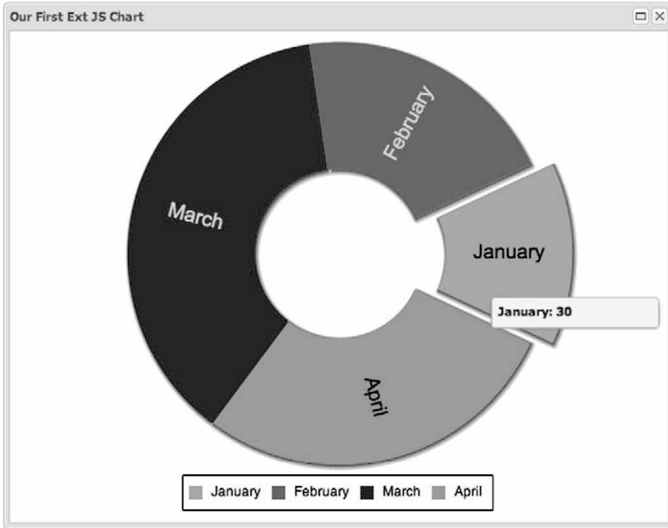


图10-19　饼图

看，并不是那么困难，它看起来很漂亮。你可以很清楚地看到，在第一学期January的数值是最小的，如果你在标签上多做一些工作，甚至可以看到是多少。我们将它留作你的家庭作业。

10.10　小结

本章介绍了绘图的基础知识。绘图是Ext JS 4引入的一个新概念，为了在浏览器中保证一致性，它经常与Flash结合使用。Ext.draw证明，通过一个简单的API，我们很容易实现优秀的、跨浏览器的绘图。

绘图是学习创建令人惊艳的图表的良好基础。不仅图表基于Ext.draw包，而且创建高级图表高度要求熟悉绘图概念。在你创建自己的第一个主题时，应该已经很清楚地了解了这一点。

本章慢慢创建一个图表，循序渐进介绍了Ext.chart图表配置背后的主要概念。你从一个小小的简单示例出发，建立了一个定制主题且具有动画的可交互图表。这些原则适用于任何序列类型。

另外，你学习了隐藏在Ext.draw和Ext.chart源码背后的很多技巧。如果你碰到什么麻烦，不要害怕深入查看这些代码。这点对于在使用Ext JS过程中碰到的所有问题都适用。

下一章是一个我们非常喜欢的主题——Ext Direct。你会通过Ext JS深入学习直接远程调用（direct remoting）技术。

第 11 章 用Ext Direct实现远程方法调用

本章内容

- 建立Ext Direct服务器和客户端
- 直接调用方法
- 通过`Ext.data.Store`使用Ext Direct

前面几章介绍了如何用网格、树形面板、表单，甚至模板（XTemplate）等部件和服务器进行数据交换，并让你使用`Ext.data.Stores`和`Ext.Ajax.request`和服务器通信。如果已经在工作中使用了一段时间的Ext JS 4，你会知道建立Ajax请求来创建、读取、更新和删除数据时，可以在配置过程中做得很精准细致，但同时也会很难处理。为每个操作的成功和失败回调操作在服务器端和客户端书写代码都将大大增加代码量，更不用提这些需要URL路径和变量来区分操作的代码的数量了。

要减少服务器端代码数量，通常的做法是调用一个RESTful接口。在客户端呢，存储和`Ext.data`包会自动完成CRUD处理的绝大多数工作，并简化开发者的工作。但是美国加利福尼亚州雷德伍德城聪明的开发者们在此基础上领先了一步。

远程过程调用（RPC），也就是远程方法调用，是一个为人所熟知的在平台间共享程序代码的技术。Ext Direct是一个将远程服务器端方法带到客户端的平台。这个灵活的、高可扩展的技术几乎与所有可以增强Web应用的服务器端平台兼容。

本身作为一层，Ext Direct被很漂亮地集成到其他框架中。虽然它的服务被很频繁地提供给数据存储（`Ext.data.proxy.Direct`）调用，但是远程可调用的方法对于直接执行也是可访问的。

本章介绍如何搭建服务器端环境并在Ext JS Web应用中充分发挥平台优势。我们还会探究Ext Direct，以便在减少数据管理方面的应用代码。

11.1 使两端相见

Ext JS 4的Ext Direct包存在于客户端，但是它依赖于服务器端的支持。客户端主要负责构建请求和处理响应。它通过多种接口来完成这些任务，例如`Ext.data.Store`或者直接方法调用。

相对于传统的通信机制而言，Ext Direct的独特之处在于服务器端。让我们从更广阔的视角看下该通信流程是怎样工作的（见图11-1）。

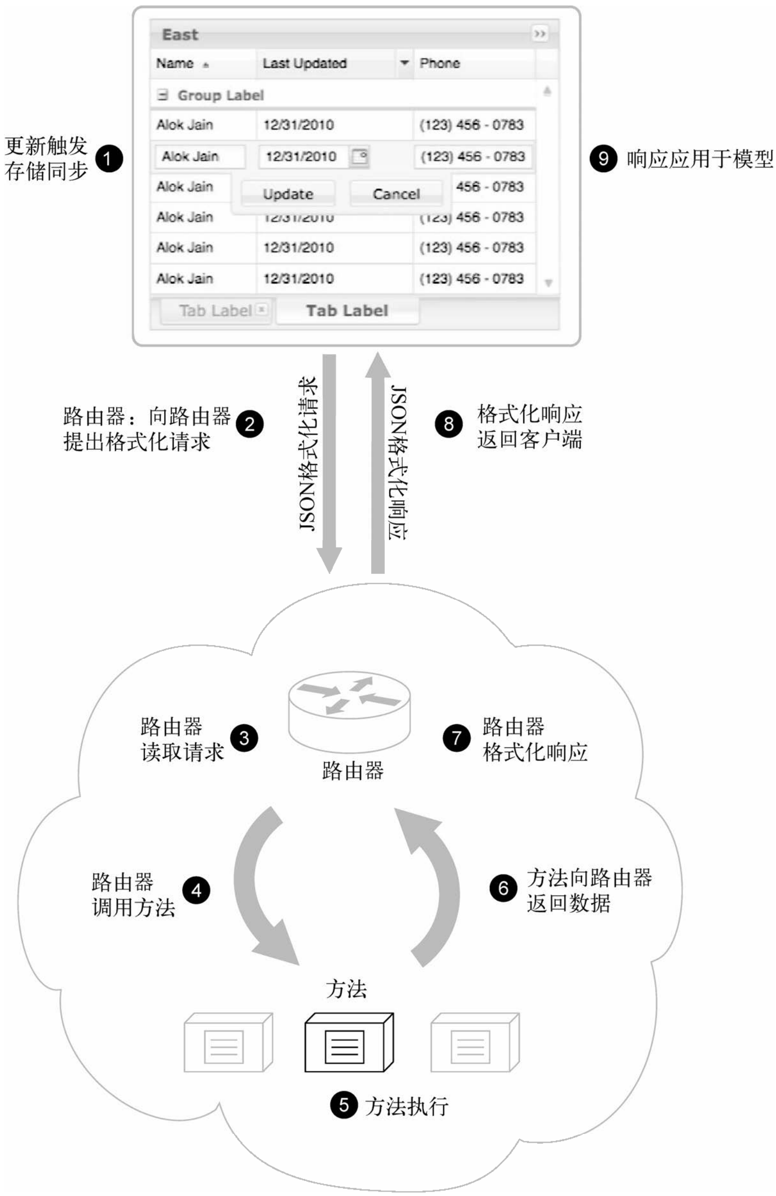


图11-1　Ext Direct的通信流程

一切都起始于Web应用客户端上的一次方法调用❶。Ext Direct将其封装到了一个JSON对象❷中，并将其发往路由器❸。路由器在该工作流中是一个特殊的机制。它知道怎样和Ext Direct

进行对话，它还知道暴露的方法在服务器端的具体位置及如何与它们进行通信。它监听来自客户端的请求，找出需要的方法并转发参数给它❹。方法❺会完成自己的工作并将值返回❻给路由器❼。该路由器会构建另外一个（与之前收到的很相似的）JSON对象❽。这个新的对象包含了远程方法❺返回的数据。最后，Ext Direct异步地收到该响应❾并通过一个回调系统将值返回给原始调用者。记住，这整个过程都是异步的。

这是过程处理的简化解释。这9步中的每一步都有更多的内容，我们将会在本章逐一探讨。这些原则展示了RPC在Ext Direct中的特殊性，强调了路由器的作用。Ext JS 4得到了其他通信类型很好的支持，其中之一就是著名的RESTful接口。让我们看看开发者是怎样权衡这两者并选择其中哪一个应用在他们的项目中。

11.2　对比 Ext Direct 和 REST

在深入了解Ext Direct之前，让我们先来从REST的视角观察RPC。很多开发者之前都使用过某种形式的RESTful接口。如果之前使用过，你可能注意到了相似点和不同点。

在Ext JS 4中，对REST和Ext Direct的支持都被推到了下一层。如果项目用数据存储来进行所有的数据管理操作，使用REST和使用Ext Direct的差别不会太明显。那是因为该框架已经使得它们相同，或者几乎相同。

不同点的存在基本上是源于每个概念的架构设计。表11-1比较了Ext Direct和REST，强调了主要的特性，这些特性是在选择将该技术用于新项目之前必须考虑的。

表11-1　Ext JS 4中Ext Direct和REST的区别

特　　性	Ext Direct	REST
服务器端配置	路由器建立和配置	类路由器行为，各种实现
客户端配置	需要为RPC提供程序的初始化提供独立的配置	不需要配置
跨域支持	不支持（基于Ajax）	支持
文件上传	通过iFrame提供支持	每个应用自己设计
批次请求（服务器端）	被路由器管理	需要被实现
写入器API	CRUD方法	CRUD路径
直接方法调用（客户端）	完全授权	非默认，方法需要手动创建

Ext Direct和REST之间最明显的差别在于它们的定义。REST以接口的形式存在，或者换句话说，以你可以提交CRUD操作的URL的形式存在。Ext Direct是一个双向平台，需要同时在存在于服务器端和客户端以使得通讯能够成功进行。它们都可以将方法暴露出来，但是Ext Direct使得服务器端方法对于客户端是可见的。更重要的是，它使得它们变得可执行。

Ext Direct通过Ajax调用来调用远程方法。唯一的例外是文件上传，它通常是使用iFrame来定期发布表单请求。这种方式的一个局限是它不具备跨域能力。因为REST是属于服务器端的接口，所以它可以被框架或浏览器中的任何有效的交流方法所使用。

批次请求是一个有趣的特性。虽然它需要路由器的支持，它的实现是集中式的，所以服务器端开发者不需要担心它。不过，REST开发者可能需要他们自己完成工作，它依赖于他们的应用。在一些情形下，每一个URL路径都有自己的批支持。在这样的情况下，Ext Direct有巨大的胜算。

在使用代理和写入器的`Ext.data`包中，设置将不会有明显的差别。你甚至可以充分利用REST的批请求，但是需要在服务器端代码中实现它们。从这点来讲Ext Direct胜出，因为对于任何调用的批支持都是被实现的，在这点上RESTful接口需要为每次调用实现批支持或者在行为前后创建一个类路由器。

Ext Direct和REST另外一个重要的不同点是在其他情形中的易用性。对于Ext Direct，它很容易通过JavaScript来调用远程方法，对于REST则需要客户创建，启用Ajax封装器来完成类似的任务。在应用中存在太多这样的代码会变成开发者的噩梦，使得应用的开发和维护变得复杂。

到目前为止你已经概览了隐藏在Ext Direct背后的理论知识，知道怎样将其和流行的RESTful接口进行比较。它们的差异之处也正是它们的优势。让我们从服务器端配置开始详细讨论。

11.3　服务器端配置

暴露服务器端方法很简单，配置起来往往也不复杂。如果你非常习惯于在服务器平台上做开发，那么尝试开发对Ext Direct的支持是个好主意。这样一来，你不仅能为自己的应用量身定制解决方案，还能更好地理解核心原则，而且会有机会享受扩展Ext Direct并增强其威力所带来的所有便利。

否则，请随意浏览并从http://mng.bz/8WkO下9种语言的30多个服务器端栈中选择，每个栈都各有特色。试用其中一些栈会让你知道如何为现有的应用选择最合适的，同时了解一些特定概念。另外，你还可能被激发自己创建栈的欲望。

11.3.1　它是怎样工作的

服务器端栈负责扮演三个角色，它需要：

- 了解哪些方法应该被暴露出去；
- 用信息为Ext Direct生成API描述；
- 监听请求，路由它们，将响应返回给客户端。

为了使Web应用可以调用到远程方法，服务器需要一个懂得两端的路由器脚本。路由器知道怎样找到你决定暴露出去的方法，并且管理两端之间的转换。

11.3.2　远程方法配置

启用Web应用RPC的第一步是让路由器知道哪些方法是有效的，怎样执行它们。路由器开发者在选择更好机制上已经完成了创造性的工作。有这样一些方法：

- 使用配置文件（非自动、最不具有创新性）；
- 要求整个类都被暴露出去，用类常量来定义怎样和类进行交流；

- 解析置于方法正上方的注释并寻找API样式的关键词；
- 使用有效方法中的完全自动测试。

根据语言的不同，一些特性可能是无效的。特定语言决定了配置方法的一些方式，并进而决定了路由器本身。路由器至少需要以下信息：方法名和期望的参数数目。额外的信息可以包含安全度量、执行前/后动作、类型转换和基于过滤器的触发器。

11.3.3 路由

路由器接收来自客户端的请求，转发合适的方法及提供的参数。客户端可以通过JSON格式的原生HTTP有效负载提交或是表单提交的方式来发送请求。需要上传文件时应选择后者，并需要分发多次请求。

每个来自客户端的事务都可能是下面这些属性的混合。

- `action`：被请求的方法所在的类。
- `method`：要执行的方法的名称。
- `data`：要传递给方法的参数。
- `type`：当前设置为`'rpc'`。
- `tid`：与该请求相关的交易ID，在批请求时是必需的。

表单提交重用字段的名字，但是添加前置关键字`ext`：

- `extAction`
- `extMethod`
- `extTID`
- `extUpload`（可选字段，用于文件上传）

任何其他的字段都可以被认为是`extMethod`的参数。

在大多数情况下，被路由器远程调用的方法都会返回数据。基于使用的语言及路由器的设计，它会捕获被调方法的输出或接受其返回的数据。如果必要的话，该数据会被处理成JSON格式，并且路由器会拼接上额外的元数据。

- `type`：设置为`'rpc'`
- `tid`：交易ID，用来识别返回数据。
- `action`：方法所在的类。
- `method`：被执行的方法的名称。
- `result`：结果数据对象的根节点。

如果该请求是被批次初始化的，它会合并结果并返回一个响应数组。如果该请求是一个表单提交和文件上传，响应将会是一个适当格式化HTML文件，该文件在文档体中只包含含有同样格式化的JSON对象的一个多行文本框。因为表单提交不支持批次，只会返回一个响应。

路由器应该捕获异常并将其以结构化的响应进行转发。启用路由器的可配置调试模式很多时候都是一个很好的主意，该模式决定了是否要传输异常。在生产环境将服务器端异常发送给客户端可以被看作是安全问题。

11

在大多数情况下，你会使用一个有效的服务器端栈，而不太关心其内部是如何处理数据的。但是有一些语言（如Node.js）并没有像在其他语言中存在的高级栈，因此你可能需要开发或者扩展相似的包（如socket.io或dnode）。

注意　在Node.js中调用远程方法是很自然的事，因为前后端使用了相同的语言，还得益于其本身架构。但是缺少一个好的服务器栈不应该让你感到沮丧。我们很确定，你能够在几个小时内创建出自己的基础路由器。《Node.js实战》（英文版由Manning于2013出版）在这方面便是一本很棒的参考书，我们强烈建议你阅读，因为Node.js还为重用JavaScript代码提供了很好的方法。

其他情况下或许需要覆盖或者扩展现有的栈，以便很好地处理MVC或者会话管理。你刚学习到的内容对于该任务来说很关键。

现在有了一个准备就绪的工作栈，请将它想象为一个足球教练。你不能没有“他”，但是还需要运动员来踢球，需要方法来执行业务逻辑。接下来，我们将向系统中添加“运动员”（即方法）。

11.4　远程方法

现在你知道了怎样使服务器端代码有效，现在是时间来使用它了。接下来创建一个简单的应用来完成下面两件事情：

- 通过一个直接方法调用来抓取一个服务器端时间戳；
- 在`Ext.data.wirter.Writer`的帮助下通过一个`Ext.grid.Panel`来启用完整的CRUD。

在该示例之后，你将使用Sencha论坛中的一个有效的服务器端栈。我们倾向于选择是出色的J.Bruni（Sencha社区的一个成员）用PHP实现的Extremely Easy Ext Direct Integration。

11.4.1　配置路由器

在配置路由器之前，请保证服务器端方法是有效的。为了简单起见，在本例中，您将创建两个PHP类，并将它们保存在一个文件里，就像下面代码清单中显示的那样。

代码清单11-1　一个简单的远程方法

```
<?php
class Util {
    public function date( $format ) {
        return date( $format );
    }
}
```

代码清单11-1新建了一个名为`Util`的PHP类。只有一个方法属于它：`date`。该方法是公共方法，这意味着它在外部是可访问的，它期望单个参数：`$format`。该方法的唯一目的就是返回

一个代表当前日期和时间的字符串，并按照传入的参数进行格式化。

该方法满足了通过一个直接方法调用在服务器端生成一个时间戳的需求。在下面的代码清单中，你将创建另外一个类来支持网格的CRUD方法。

代码清单11-2　一个远程CRUD类

```
class Actors {
    public function create($config) {
        return Array(
            "success"=> true,
            "data"=>Array("name"=>"New Actor", "id" => rand(1,22000))
        );
    }

    public function read( $config ) {
        return Array("success"=> true, "data"=>Array(
                Array(
                     "id"   => rand(1,22000),
                     "name"   => "John Travolta"
                ),
                Array(
                     "id"     => rand(1,22000),
                     "name"   => "Benny Hill"
                ),
                Array(
                     "id"     => rand(1,22000),
                     "name"   => "Bruce Willis"
                ),
                Array(
                     "id"     => rand(1,22000),
                     "name"   => "Rowan Atkinson"
                )
          ));
    }

    public function update( $config ) {
        return Array("success"=>true, "data"=>$config);
    }

    public function destroy( $config ) {
        return Array("success"=>true);
    }
}
```


代码清单11-2新建了一个类，它包含4个公共方法：`create`、`read`、`update`、`destroy`。它们均不做什么实际工作，只是向客户端返回最少的需求信息。实际上，这4个方法在需要的时候会做验证，与数据库交互，然后返回来自数据库的响应以完成该业务。

如果已经下载了该PHP路由器，现在是时间将其插入了。你会将该服务器端栈API包括在新创建的路由器中并配置它，就像下面的代码清单显示的那样。

代码清单11-3　Router配置

```
<?
require 'ExtDirect.php';

ExtDirect::$namespace = 'RPC';
ExtDirect::$descriptor = 'RPC.REMOTING_API';
ExtDirect::$enableBuffer = 200;
```

这个路由器的配置相当简单：返回rpc.php文件的开始，并将ExtDirect.php文件的引用追加进去。你是对的，ExtDirect.php是该栈的路由器文件。将该路由器文件包含到rpc.php文件中可以保证该路由器的API是有效可用的。

路由器只有很少的几个配置选项。但是此时此刻只需配置要在客户端使用的命名空间和Ext Direct描述符。后者将被用来通知你已经使之生效了的方法的Ext Direct。

因为你想使应用变得更加智能并通过批请求来重用资源，所以会启用缓存并将其设置为200 ms。

现在已经成功配置好了RPC服务器端路由器。该独有的安装将会明白所有的公共方法都会被置为有效的，并收集需要的参数的个数，而且会在Web应用请求它的时候重新配置该API描述。

旅程继续进行到客户端。Ext Direct需要知道怎样执行操作、路由器在哪里、可以远程调用哪些方法。它会使用到你刚才已经探索过的配置，详情我们稍后介绍。

11.4.2　启用Ext Direct

`Ext.direct.Manager`主管客户端的Ext Direct。它的工作是为要使用的每个提供程序（provider）创建、缓存和管理实例。现在存在两个不同的提供程序：`PollingProvider`和`RemotingProvider`。

`Ext.direct.PollingProvider`有一个责任：在可配置的时间间隔执行一个简单的远程方法。它还按照预定义的规则（在一个回调函数中指定）来处理信息。一个示例情景是登入服务器使其知道一个用户依然在线，并在同时刷新需要传送给用户的信息的可见性。

另一方面，`Ext.direct.RemotingProvider`根据需要调用远程方法。它通常用来和`Ext.data.Store`相连，包括读取器和写入器。它的使用模式和`PollingProvider`很相似，我们会集中在另一个示例中讲述。

让我们来看下图11-2中两个提供程序的主要区别。最显著地，`PollingProvider`在给定的时间间隔重复一个请求。它将会重复这种方式直到被强制停止。`RemotingProvider`在客户端创建了一个远程方法的别名。只有当客户端方法被调用了，Ext Direct才会发送一个请求道服务器端并通过回调函数来处理响应。相同的客户端方法可以被手动执行（以编程方式），或者通过`Ext.data.Store.load()`方法或`sync()`方法执行。

现在已经指定了该远程提供程序，接下来通过`Ext.direct.Manager`抓取API描述并处理它。为此，添加一个额外的`<script>`标签，就放在该框架的引用后面：

```
<script type="text/javascript" src="rpc.php?javascript"></script>
```

现在是时候来看下rpc.php中的示例代码了。你可以在examples/ch11/direct_app/rpc.php中找到它。

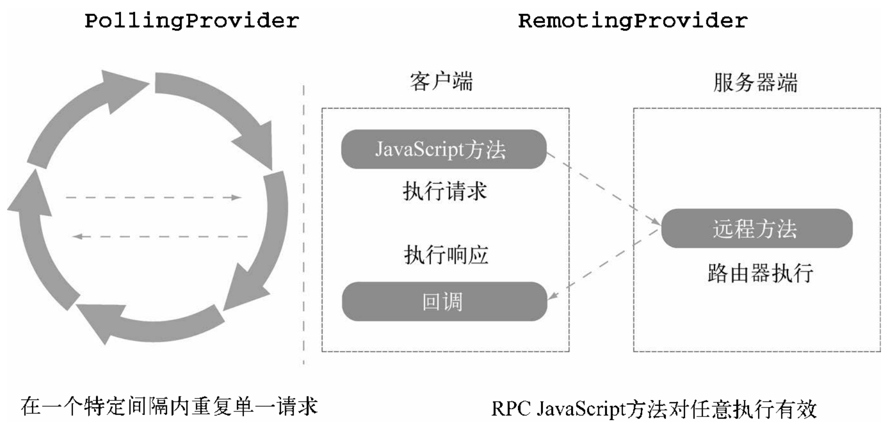


图11-2　PollingProvider与RemotingProvider的对比

该路由器会自动返回一个配置，并通过在获取到的JavaScript文件尾部拼接一个特殊行来创建Provider的一个实例：

```
Ext.Direct.addProvider(RPC.REMOTING_API);
```

在这步，Ext Direct将会自动初始化该namespace并create方法。然后所有的远程有效方法将会通过之前配置的命名空间来访问。Util.date变成RPC.Util.date，Actor.read变成RPC.Actor.read。

> **在初始化一个提供程序之前额外的JavaScript调用**
>
> 如果项目使用Ext.Loader，在通过添加Ext.syncRequire('Ext.direct.Manager')调用Ext.Direct.addProvider之前需要Ext.direct.Manager。此外，如果应用之前没有被定义过，你需要通过Ext.ns('RPC')定义它，RPC是定义在API描述中的命名空间。

看一下代码清单11-4，它显示了一个API描述的示例。

代码清单11-4　示例API描述

```
RPC.REMOTING_API = {
    "url"           :"\/extdirect\/rpc.php",
    "type"          :"remoting",
    "namespace"     :"RPC",
    "descriptor"    :"RPC.REMOTING_API",
    "enableBuffer"  :1000,
    "actions":{
        "Actors":[
            {"name":"create",   "len":1},
            {"name":"read",     "len":1},
            {"name":"update",   "len":1},
            {"name":"destroy",  "len":1}
```

```
            ],
              "Util":[
                {"name":"date",      "len":1}
              ]
          }
      };
```

服务器端返回了一个变量，该变量将在执行过程中被Ext Direct所知。该变量将引用一个最少包含三个参数的对象：

- ❑ 到路由器的URL❶；
- ❑ 服务类型（远程/轮询）❷；
- ❑ 动作、代表类、方法，以及每个方法接受的参数个数❺。

其他都被计入特殊配置项。因为你不应该污染一个全局命名空间，所以将它配置为了RPC❸。你可以为Ext Direct选择应用的命名空间或一些特殊的东西——这基于开发者的偏爱。你还选择了启用批缓存并将其设置为1000 ms（1 s）❹。这意味着Ext Direct会将第一个请求延迟1秒并等待其他连续的调用，以将其作为批处理为一个请求发送出去。这样一个特性需要路由器的支持，因为这些请求会被组合为每项都是唯一标识调用的数组。

下面的代码清单包含了一个为Sencha Architect用户实现的示例API描述配置对象。

代码清单11-5　为SenchaArchitect配置的示例API描述

```
{
    "url"            :"\/extdirect\/rpc.php",
    "type"           :"remoting",
    "namespace"      :"RPC",
    "descriptor"     :"RPC.REMOTING_API",          <─┐
    "enableBuffer"   :1000,                          ❶ 指定API 描述符名
    "actions":{
        "Actors":[
            {"name":"create",    "len":1},
            {"name":"read",      "len":1},
            {"name":"update",    "len":1},
            {"name":"destroy",   "len":1}
        ],
        "Util":[
            {"name":"date",      "len":1}
        ]
    }
}
```

Sencha Architect的用户会很欣赏Architect为进一步使用存储和部件而导入API描述的能力。主要的差异在于该描述需要是JSON格式的，并包含该API 描述符的名字❶。除了格式发生了变化，其内容和之前代码清单中的完全相同。

一个重要的里程碑达成了：所有的准备工作都已经准备就绪。你已经创建了一个路由器，描述了API并将其包含在了Web应用中。Ext Direct已经创建了一个提供程序实例，现在可以使用远程方法了。在本章稍后的内容中，你会看到在一个Ext JS应用使用两种主要的RPC：通过JavaScript的直接方法调用和通过`Ext.data.Store`的CRUD操作。

11.5　直接调用远程方法

就像我们在本章早些时候提到的那样，Ext Direct通过解决`Ext.Ajax`请求来处理调用。这意味着该处理是异步的；为了对响应数据做一些操作，回调函数是必需的。每个客户端远程方法和远程副本拥有相同的参数数目，还有回调和作用域。在下面的代码清单中，你将会使用`RPC.Util.date`。

代码清单11-6　直接调用一个远程方法

```
var callbackFn = function(res) {
    this.log(res);
    this.timeEnd('DirectTiming');
}

console.time('DirectTiming');
RPC.Util.date('d/m/Y', callbackFn, console);
```

在`callbackFn`中，你只将最终结果输出到控制台。为了演示往返速度，这里需定义一个新的控制台计时器，只有在回调函数被调用时该计时器才会被终止。现在回调函数已经准备好了，控制台也已经可以用于检测，现在是时候执行`RPC.Util.date`了。日期模式`'d/m/Y'`被发送给路由器并进而转发给目标方法，该方法会将格式化的日期返回给路由器。数据被封装在一个JSON对象中并返回给浏览器。Ext Direct接收到该数据，通过`tid`识别它，然后将该数据作为第一参数传递给回调函数并执行它：

```
25/01/2013
DirectTiming: 151ms
```

`RPC.Util.date`很好地在浏览器的控制台上输出了两行：日期，完全以你想要的格式呈现；完成整个操作所需要的时间。该测试从一个远程位置执行，使用了一个移动网络。

到目前为止还不错。现在让我们增添点乐趣，使用下面代码清单中的联系请求。你将用不同的参数三次调用相同的方法。观察该网页检测器的网络标签会特别有趣。

代码清单11-7　批次请求

```
RPC.Util.date('d/m/Y', console.log, console);
RPC.Util.date('H:i', console.log, console);
RPC.Util.date('U', console.log, console);
```

所有的请求几乎都在同一时间发送。你将会注意到回调函数是一个简单的控制台日志，也就是说它会将返回结果输出到控制台。就像预期的那样，所有的结果在同一时间被返回，在远程处理时间和往返时间之后的1 s。

但是，Ext Direct怎样区别来自客户端和服务器端的请求呢？让我们看看下面的代码清单。

代码清单11-8　批次请求负载

```
[
    {
```

```
            "action":"Util",
            "method":"date",
            "data":["d/m/Y"],
            "type":"rpc",
            "tid":2
        },
        {
            "action":"Util",
            "method":"date",
            "data":["H:i"],
            "type":"rpc",
            "tid":3
        },
        {
            "action":"Util",
            "method":"date",
            "data":["U"],
            "type":"rpc",
            "tid":4
        }
    ]
```

一个简单请求和批请求之间唯一重要的区别是：后者以一个请求对象的数组的形式被发送。每个请求对象都用一个`tid`来标识，它可以被用来区别服务器端的所有调用。该路由器还将需要为一个请求的数组返回相同的ID，我们很快会回来探讨。

被用于远程方法的参数是以数组的形式被发送的。未来，Sencha可能会支持键值对（key/value），但是现在请确认所有的参数是按照正确顺序调用的。

每一次请求都应该有一个恰当的响应，让我们在下面的代码清单中看一下。

代码清单11-9　批次响应

```
    [
        {
            "type":"rpc",
            "tid":2,
            "action":"Util",
            "method":"date",
            "result":"27\/01\/2013"
        },
        {
            "type":"rpc",
            "tid":3,
            "action":"Util",
            "method":"date",
            "result":"03:08"
        },
        {
            "type":"rpc",
```

```
            "tid":4,
            "action":"Util",
            "method":"date",
            "result":"1327633709"
        }
    ]
```

请求和响应看起来像“双胞胎”，但是它们之间有一个重要区别。虽然在请求中参数对应`data`键，但是返回值现在匹配的是`result`键。请记住响应也是一个对象数组，这些对象都包含`tid`属性来匹配到合适的请求。

你已经知道怎样直接执行、调试甚至检测RPC，且会经常通过`Ext.data.Store`使用Ext Direct以使用部件中的数据。让我们来看下Ext Direct是怎样使书写代码变得更加简单和快捷的。你还会使用Ext Direct在一个`Ext.grid.Panel`中实现完整的CRUD处理。开始吧！

11.6 启用 CRUD 的 `Ext.data.DirectStore`

接下来创建一个可编辑的`Ext.grid.Panel`的实例（参见图11-3），并使用`Ext.data.Store`中的写入器配置来利用远程端的自动数据同步。看起来有好多事情要做，但是这样的工作对于Ext JS 4和Ext Direct来说只是小菜一碟儿。

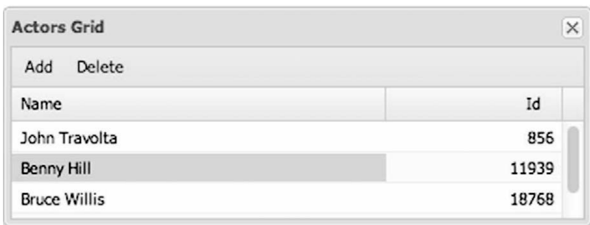


图11-3 与远程端进行自动同步数据的可编辑网格

这个示例可以和你之前使用过的第一个示例共用一个index.html文件和相同的API 描述符。这意味着你已经知道了怎样初始化Ext Direct并让其知道哪些方法是有效的。还有，再看一下代码清单11-2，它用PHP为第二个示例创建了一些方法。

首先解决最困难的部分：`Ext.data.Model`，如下面的代码清单显示。`Ext.data. Model`负责任何数据特定的事情：从服务器抓取记录、创建新的记录、更新和删除、调用Ext Direct方法以及执行回调。

代码清单11-10 配置`Ext.data.Model`

```
Ext.define('Actor', {
    extend  : 'Ext.data.Model',

    fields  : [
        'name',
        'data'
```

```
    ],
     proxy: {
        type : 'direct',                    ❶ 添加Direct代理
           api  : {
               create   : RPC.Actors.create,        ❷ 设置CRUD API方法
               read     : RPC.Actors.read,
               update   : RPC.Actors.update,
               destroy  : RPC.Actors.destroy
        },
        writer  : {
            type: 'json',
            writeAllFields: true
        },
        reader  : {
            root             : 'data',
            idProperty       : 'id',
            type             : 'json',
            successProperty  : 'success'
        }
    }
});
```

你已经处理过存储、代理、读取器和写入器，所以对于该示例的绝大部分应该都可以从容应对。两个不寻常的配置项是代理类型❶和CRUD API❷。在API的配置属性中，你已经告诉了代理对于每个CRUD操作应该使用哪个方法。这是对Ext Direct应该做的所有设置。现在必须创建一些处理程序来添加和删除数据，就像下面代码清单中显示的那样。

代码清单11-11 支持处理程序和实例

```
var editing = Ext.create('Ext.grid.plugin.CellEditing'),    ❶ 实例化CellEditing插件
     grid,
    onAdd,
    onDelete;

     onAdd = function() {                            ❷ 添加新记录
          var record = Ext.create('Actor');
          editing.cancelEdit();
          grid.getStore().insert(0, record);
          editing.startEditByPosition({
               row: 0,
               column: 0
          });
}
                                                     ❸ 删除记录
onDelete = function(){
     var view      = grid.getView(),
         selection = view.getSelectionModel().getSelection()[0];
     if (selection) {
          grid.getStore().remove(selection);
     }
}
```

第一步是实例化CellEditing插件❶，你选择它作为对网格的编辑选项，使用了该网格的引用，

然后为工具栏按钮设置了onAdd❷和onDelete❸处理程序。这两个处理程序都将直接处理存储，但不会以任何形式和Ext Direct一起工作。Ext.data.Store将完成这项工作。

注意 你没有污染全局的命名空间，因为已经在Ext.onReady中包含了所有代码。

最后创建该网格，如代码清单11-12所示。这将会很有趣！

代码清单11-12 网格配置

```
grid = Ext.create('Ext.grid.Panel', {
        height      : 350,
        width       : 600,
        title       : 'Actors Grid',
        renderTo    : Ext.getBody(),
        selType     : 'cellmodel',
        store       : {
            model   : 'Actor',
            autoLoad: true,
          autoSync: true                         ❶ 与服务器同步数据
        },
        columns     : [{
            dataIndex   : 'name',
            flex        : 1,
            text        : 'Name',
               field       : {
                    type     : 'textfield'
               }
        }, {
            dataIndex   : 'id',
            align       : 'right',
            width       : 120,
            text        : 'Id'
        }],
          plugins     : [
               editing
          ],
          dockedItems : [
               {
                    xtype    : 'toolbar',
                    dock     : 'top',
                    items: [
                         {
                              text     : 'Add',
                              handler : onAdd              ❷ 指派onAdd处理程序
                         },
                         {
                              text     : 'Delete',
                              handler : onDelete           ❸ 指派onDelete处理程序
                         }
                    ]
               }
```

```
        ]
    });
```

关键的配置项是存储设置❶。你设置`autoSync`为`true`，这意味着`Ext.data.Store`将会监控数据变化并将它们推送到服务器。有了这个设置，网格就完成了，并被如图11-4所示那样渲染。

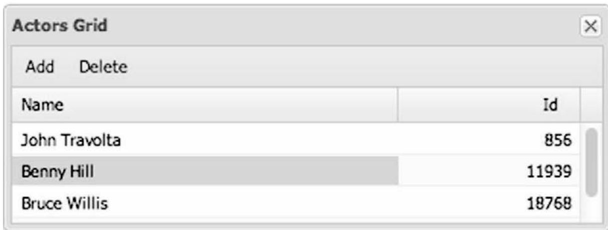


图11-4　可运行的使用Ext Direct的数据网格

对于每个响应都应该返回一个带有`success：true`属性的对象以使存储知道一起正常。否则，你便给出了失败的描述。对于读取、更新和销毁都是这样的，然而添加数据将会有额外的信息被返回：ID和新的记录。一旦用户点击了Add按钮❷，存储会创建一个新的记录并和服务器端确认。用户将继续编辑该条新的记录，`Ext.data.Store`则更新该条记录的ID（一旦收到它）。更改应该在同一时刻发生，存储会将为保存的字段标记一个红色三角，并等待下一次同步再发送这些更改。

在不超过100行的代码里，你已经创建了一个网格，它从一台服务器读取数据，还允许创建、编辑和删除记录❸。对于那些快速键入，它还可以缓存请求，可以在短时间内修改一些列。这将极大地节省带宽，并且有时会助升性能。

11.7　小结

Ext Direct是一个用来调用远程方法的强大机制。它的主要目的是简化服务器端和客户端的通信，同时减少实现该项工作的代码。是的，有一些API（比如dnode）做相似的工作，但是没有一个和Ext JS的数据存储集成得如此漂亮，为新项目和现有项目提供完整的CRUD工作流。简单的范围控制是该包的另一个独特优势。

现在你已经看到了怎样在服务器端和客户端之间传输数据，让我们继续在用户界面元素间交互性地传输数据。下一章将为应用添加拖放功能。

第 12 章 拖放

本章内容

- ❑ 理解拖放工作流
- ❑ 剖析Ext JS拖放类
- ❑ 实现拖放的覆盖方法
- ❑ 理解拖放的生命周期
- ❑ 使用拖放插件

使用用户图形界面最大的一个好处就是：能够在界面上通过简易的鼠标动作移动元素。这种用户交互称作拖放。每一次使用现代计算机，你几乎都会毫不犹豫地使用拖放功能，这个功能极大地方便了人们的生活。

想要删除一个文件，点击并拖动文件图标把它放到垃圾箱或者回收站图标上即可。很容易，是不是？如果没有拖放功能呢？怎样才能把文件从它所在的位置放到回收站中呢？让我们想想有哪些可能：可以首先点击该文件并聚焦，然后通过一个组合键来剪切文件，再找到并聚焦回收站窗口，最后通过一个组合键把文件粘贴进去。另外一种做法是点击选中文件并聚焦，按下键盘上的删除键（Delete），但如果你是右撇子，这样做需要把手从鼠标上挪开。相反，拖放就简单多了，不是吗？现在想象自己是富网络应用的用户，怎样通过拖放来简化用户体验呢？

幸运的是，Ext JS提供了拖放功能。在本章中你将看到，只要努力和下决心就可以在应用中加入拖放功能。首先是在基本DOM元素上添加拖放功能，这将为在部件（如网格和树形面板）上添加这些行为打下基础。

12.1 拖放工作流

拖放功能发生的前提是计算机知道哪些东西是可以或者不可以被拖动的，还要知道哪些东西可以或者不可以被放入。比如说，桌面上的图标通常是可以被拖动的，但是其他一些东西，比如说Windows任务栏上的时钟或者Mac OS X等上的菜单栏则不可以。这一级别的控制对于允许某些工作流的执行是必要的，我们将会讨论这一点。

为了有效地利用拖放，你需要理解完整的工作流。本节把工作流按照拖放的生命周期划分成三个主要的部分：拖动的开始、拖动动作和放入。

12.1.1　拖放的生命周期

以电脑桌面作为示范，桌面上的任何图标都可以被拖动，但是只有一小部分可以被放入，通常是磁盘或者文件夹图标，垃圾回收站或者是可执行程序图标。对于Ext JS来说，拖放同样需要这样的注册机制。任何一个元素都需要像这样被初始化才能够进行拖放。要让DOM元素可以被拖放，它至少必须注册成拖动项或是放入项。一旦元素被这样注册了，拖放就可以起作用了。

拖动操作是开始于通过鼠标点击并点住电脑图标，接着按住鼠标并拖动。电脑会基于前面描述的注册机制来判断一个项目是否可以被拖动的。如果不能，那么什么都不会发生。用户可以点击并尝试拖动操作，但是什么都不会发生。要是一个元素是允许被拖动的，图形界面通常就会创建该拖动项的一个轻量级的复制品，也叫作*拖动代理*，它会跟随鼠标广标的移动。这给用户感觉他们真的是在屏幕上拖动那些东西。

对于在拖动动作过程中鼠标光标的每一个变化（或者说*X*-*Y*坐标轴的变化），电脑都会判断是否可以把拖动项放入光标所在的位置。如果是可以被放入的位置，就会出现某种可视化的放入操作引导。如图12-1所示，在文件图标代理被拖过文件图标上时，你就会看到一个放入引导。

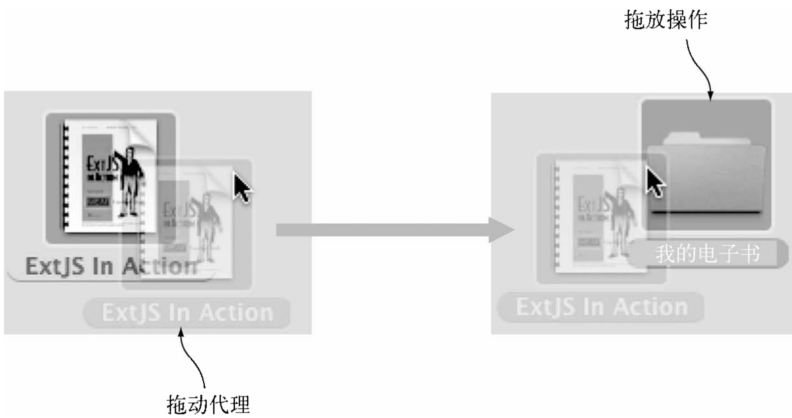


图12-1　Mac OS X 操作系统上的桌面拖放交互，左边是由拖动事件触发而创建的拖动代理，右边则展示了放入引导

拖放的生命周期在放入动作发生之后就结束了，也就是在拖动操作中鼠标按键释放之后。此时，计算机必须决定伴随放入操作要做的事情。放入事件是否发生在一个合法的放入目标上？如果是，那么放入目标项是不是和拖动项属于同一个拖放组呢？是把拖动项进行复制还是移动呢？这样的决定大多是由应用程序的逻辑自己来决定的，这也是最需要写代码的地方。

虽然解释拖放行为相对容易，但是实现起来却相当地困难，虽然不是不可能。要想高效地实现拖放的一个关键要素是对于类层次以及每个类应该做什么有基本的理解。这一点对于在基础

DOM级别上实现拖放功能是正确的，并且你也将会看到这一点是在Ext JS UI部件的拖放功能的实现过程中体现。

出发啦！我们先爬上三万英尺并鸟瞰拖放的类文件的层次结构。

12.1.2　自上而下审视拖放类

乍一看，拖放类列表可能有点儿让人招架不住。当Jesus第一次看到API文档里面的类列表，被这些选项吓了一跳。这个所谓的框架有11个类，可以提供基本功能（比如实现可拖放的DOM元素），也可以提供复杂的功能（比如可以通过代理来拖放多个节点）。最酷的是，一旦你从上而下地审视这些类，就会发现组织这些支持类并理解它们的作用并非难事。

我们从这里开始探索。图12-2展现了类层次结构，其中有11个拖放类。Ext JS框架的所有拖放功能都是从`DragDrop`类开始的。`DragDrop`类提供了拖放功能需要的所有基本方法，这些方法就是用来被覆盖以供使用的。它仅仅是提供了实现拖放行为的最基本的工具，你要负责写代码来完成整个实现。

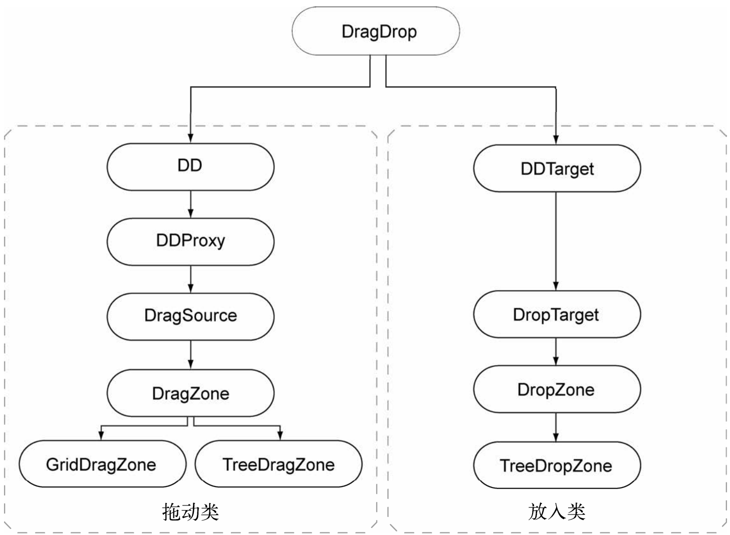


图12-2　拖放类层次结构可以划分成两个主要部分：拖动（左）、放入（右）

这是理解拖放工作原理的关键，因为这个设计模式贯穿了整个拖放类层次结构。这种理念是非常强大的，因为当你拥有了可以给应用程序添加行为的基本工具类，就会很容易确保拖放功能的正常工作。

从上往下看类的继承链，就会看到一个分叉，左边是从`DD`开始，右边从`DDTarget`开始。`DD`是所有拖动操作的基类，`DDTarget`是所有放入操作的基类。它们两个都分别提供了相关行为的基本功能。这一功能上的解耦让你可以关注特定的行为。待会儿我们研究怎么在DOM节点上实现拖放功能的时候，你就会看到这一点是如何产生作用的了。

继续往下看继承链，你就会看到Ext JS在为预期的行为渐进式增加功能。表12-1罗列了所有的类以及对它们的设计初衷的简单描述。

表12-1　拖放类

拖动类		放入类	
名　称	用　途	名　称	用　途
DD	基本的拖动实现，元素可以被拖动或者放入到任何地方。这是DOM级别的拖放实现最多用到的类	DDTarget	允许任意元素参与到拖放组的基本类，但是它不可以被拖动，也就是说它只可以被放入元素
DDProxy	基本拖动的实现，用来实例化拖动元素的一个轻量级实现，也称作拖动代理，它替代了源元素来被拖动。在需要拖动代理的时候使用这个类是惯用的手法	DropTarget	该基类提供了一个放入类的空管道，它会在可拖动元素被放入进来的时候起作用。开发人员负责通过覆盖notify方法来完成功能
DragSource	提供了使用状态代理拖放的基本实现，也是DragZone的基类。它可以被直接使用，但是更常用的是DragZone类（下一项）	DropZone	提供多种节点可以被放入的类，它和DragZone类一起使用最佳。Ext.Grid.header.DropZone是专门为网格面板特别实现的DropZone。Ext.tree.View DropZone则是为树形面板特别实现的
DragZone	这个类允许同时拖动多个DOM元素，并且经常和视图部件一起使用。为了给网格面板和树形面板提供拖放功能，它们各自有这个类的实现，分别是Ext.grid.header.DragZone和Ext.tree.ViewDragZone		

你都看到了：每一个拖放类和它们的设计初衷。一个拖动项（DD类或者其子类）可以是放入目标，但是一个放入目标（DDTarget类或者其子类）却不能是一个拖动项。了解这一点是很重要的，因为当你决定要有一个既能作为拖动项又要作为放入目标的元素时，就必须使用一个拖动类了。

如果是在通用DOM节点上实现拖放功能，并且需要一次仅允许拖动一个节点，可以使用DD类或者DDProxy类。如果拖动超过一个元素，需要使用DragSource类或者DragZone类。这就是为什么树形面板和网格面板拥有它们自己的DragZone类的扩展或实现。

同样，如果想要放入单个节点，那么DDTarget类就将是要选择的放入类。对于多节点的放入操作就需要DropTarget类或者DropZone类了，因为这两个类是和DragSource、DragZone以及它们的子类交互的必要渠道。

知道这些类是什么是拼图的一部分。下一部分你需要了解哪些方法需要被覆盖。这是成功部署的最关键部分。

12.1.3　一切尽在覆盖之中

正如我们之前所说的，多种多样的拖放类只是用来提供一个支持拖放行为的基础框架，仅仅是完成拖放功能并使之可用的一部分。每一个拖放类都包含一系列的抽象方法，需要由你（终端

应用程序员）来覆盖（见表12-2）。

表12-2 拖放功能常用的抽象方法

方 法	说 明
onDrag	当元素被拖动触发onMouseMove事件时调用。如果想要在拖动之前做些操作，可能要选择覆盖方法b4Drag或者startDrag
onDragEnter	在拖动元素第一次碰到同一个拖放组内的另一个拖放元素时调用。这里可以给放入引导写代码
onDragOver	在一个拖动元素被拖动出来的时候调用
onDragOut	在拖动元素离开了相关的拖动或放入元素占用的物理空间后调用
onValidDrop	在拖动元素被放入到任意的不相关拖动或放入元素上时调用。这是为告诉用户他们在错误的地方放入了拖动元素而注入提示的绝佳之处
onDragDrop	在拖动元素被放入到同一个拖放组内另一个拖放元素上时调用

虽然所有这些方法都在框架API文档的拖放章节罗列了，但是最好还是简单介绍一下一部分Ext.dd.DD类上非常常用并且需要被覆盖的抽象方法。这样的话，你就能意识到自己应该重点看API文档中的哪一部分了。

记住，Ext.dd.DD类是所有拖动相关的元素的基类，当你从上而下地审视类层次时，会发现有不少功能被逐渐添加上去。由这些子类增加的功能会逐步覆盖那些方法。

举个列子，Ext.dd.DD提供了一些b4（before）方法，让你可以编写自己的行为让它在某事发生之前执行。其中有在鼠标按下事件之前执行（b4MouseDown），也有在拖动一个元素之前执行（b4StartDrag）。Ext.dd.DDProxy类是Ext.dd.DD类的第一个子类。它覆盖了用来创建可拖动代理的方法，该类方法会在拖动代码执行之前被调用。

要想知道覆盖哪些方法以达到特定的实现，需要向API去请教那个特定的拖放类。因为Ext JS的更新速度周期非常快，一些方法可能被添加，重命名或者删除，所以定期地看API文档会帮助你了解最新的变化。

对于拖放工作原理最后一点我们需要加以讨论的是拖放组的使用，以及它在开发应用程序中意味着什么。

12.1.4 拖放总是在组中工作的

拖放元素是和组相关联的。组是一个基础的约束条件，用来管理一个拖动元素是否可以被放入到另外一个元素上。组是一个标签帮助拖放框架来决定一个注册了的拖动元素是否可以和另外一个注册了的拖动或是放入元素交互。

拖放元素必须关联到至少一个组，甚至可以关联到多个。它们通常在初始化的时候就和一个组相关联并且通过addToGroup方法和更多的组相关联。同样的，它们也可以通过removeFromGroup方法解除关联。

这是理解Ext JS拖放模块基础的最后一块拼图了。现在是时候开始使用并强化所学知识的了。我们从开发DOM元素的拖放功能开始。

12.2　拖放：一个基础的例子

在探索之前，我们先来模拟游泳池的设置，它有更衣室、泳池以及热水浴缸。在这里，你需要遵守一些约束条件。比如说，男士和女士只能够根据性别进入对应的更衣室。所有人都可以进入游泳池，但是只有一部分喜欢进入热水浴缸。现在知道需要构建什么了，开始写代码吧。

正如你即将看到的，配置一个可以在屏幕上拖动的元素极其简单，但在此之前必须先创建一个可操作的工作区。可以通过创建一些CSS样式来管理一组特定的DOM元素的样式，然后把拖动逻辑应用到它们身上。我们会使事情尽可能得简单，从而把注意力放在主要问题上。

12.2.1　创建一个小型工作区

如下面的代码清单所示，首先要创建用来代表更衣室和进入其中的人的标记。这个代码清单蛮长的，因为HTML需要建立所要求的样式和布局。

代码清单12-1　创建拖放工作区

```
<style type="text/css">
     body  {
         padding: 10px;                              ❶ 配置拖放元素容器样式
     }
     .lockerRoom {
         width:              150px;
         border:             1px solid;
         padding:            10px;
         background-color: #ECECEC;
     }
     .lockerRoom div {                               ❷ 使子节点看起来不同
         border:              1px solid #FF0000;
         background-color:   #FFFFFF;
         padding:             2px;
         margin:              5px;
         cursor:              move;
     }
</style>
<table>
    <tr>
         <td align='center'>
             Male Locker Room
         </td>
         <td align='center'>
             Female Locker Room
         </td>
    </tr>
    <tr>                                              ❸ 为我们的拖放元素
        <td>                                             设置HTML
            <div id="maleLockerRoom" class="lockerRoom">
                <div>Jack</div>
                <div>Aaron</div>
                <div>Abe</div>
```

```
            </div>

        </td>
        <td>
            <div id="femaleLockerRoom" class="lockerRoom">
                <div>Sara</div>
                <div>Jill</div>
                <div>Betsy</div>
            </div>
        </td>
    </tr>
</table>
```

代码清单12-1创建了CSS样式和标记用来为探索基本DOM元素的拖放创造了条件。从定义CSS样式开始，CSS会控制`lockerRoom`❶元素容器以及它们的子节点的样式❷。然后可以设置使用这些CSS的标记❸。

图12-3展现了渲染在屏幕上的更衣室HTML页面。注意，当把光标移动到更衣室元素的子节点上时，光标就会变成一个十字。这源自于前面配置的CSS样式，这是提示拖动动作的好方法。

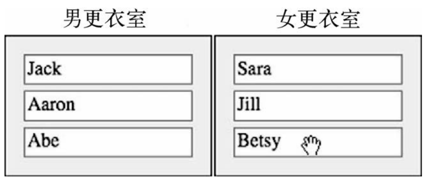


图12-3　渲染在屏幕上的更衣室HTML代码

接下来需要配置 JavaScript 使这些元素可以被拖动。

12.2.2　配置元素使之可拖动

代码清单12-2配置更衣室子元素，使之可以在屏幕上被拖动。要完成这一目标，需要在`Ext.get`调用的结果上执行`select`调用把`maleLockerRoom`元素里的子元素聚集起来。然后利用`Ext.each`来遍历整个子节点的链表并创建一个新的`Ext.dd.DD`的实例并传入`element`的引用，这样就使得元素可以在屏幕上被拖动了。可以对`femaleLockerRoom`里面的元素做同样的事情。

代码清单12-2　启用元素的拖动功能

```
var maleElements = Ext.get('maleLockerRoom').select('div');
Ext.each(maleElements.elements, function(el) {
    new Ext.dd.DD(el);
});
var femaleElements = Ext.get('femaleLockerRoom').select('div');
Ext.each(femaleElements.elements, function(el) {
    new Ext.dd.DD(el);
});
```

刷新页面之后就可以在屏幕上轻松地拖放元素了。如图12-4所示，可以不受约束地在屏幕上拖放任意的子div元素了。

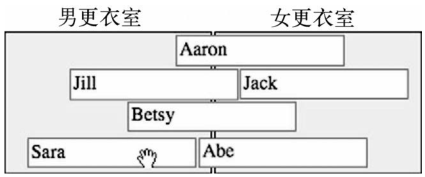


图12-4　在更衣室上允许拖动操作了，并且不受任何约束

让我们仔细观察下Ext.dd.DD是如何工作的，以及它对DOM元素做了什么。为此，需要刷新页面，并打开Firebug的在线HTML检查工具。我们将集中看Jack元素。

12.2.3　分析Ext.dd.DD的DOM元素变化

图12-5通过Firebug提供的DOM检查工具展示了页面更新之后瞬间拖动元素的HTML代码。



图12-5　检查拖动操作发生前的DOM元素Jack（高亮）

当着眼于Jack元素（图12-5中的高亮部分）的时候，首先映入眼帘的是它被赋予了唯一的ID，名为"ext-gen3"。注意，在标记文本中并没有给这个元素赋予一个ID。如果一个元素已经有一个自己的ID，Ext.dd.DD就会使用它。但是为了通过ID跟踪这个元素，这个元素被Ext.dd.DD的超类Ext.dd.DragDrop赋予了一个ID。

警告　如果一个元素的id在注册为拖动元素之后改变了，那么这个元素的拖动配置也就随之失效了。如果打算更改一个特定元素的id值，最好的办法是调用这个元素的Ext.dd.DD实例上的destroy方法，并创建一个新的Ext.dd.DD实例，同时传入一个新的ID值作为第一个参数。

你可能还在HTML检查工具上注意到了另外一点，这个元素上没有被赋予其他的属性。如图12-6所示，现在可以稍微拖动一下元素，并观察其中的变化。



图12-6 观察拖动操作给Jack元素带来的变化

可以看到，已经把Jack元素拖动了一点点。Ext.dd.DD则依次地把样式属性加到元素上，这些样式会改变position、top和left的CSS特征。明白了这一点是很重要的，因为使用Ext.dd.DD会导致屏幕上的元素的位置变化，这一点和我们即将要介绍的Ext.dd.DDProxy是极为不同的。

最后我们要讨论的是拖动元素如何被看上去放入到某个地方。乍看之下，这极其酷炫，确实如此，但是其实并没有什么用。要想让它有用，需要给它加上约束条件。

要达到这一目标，需要创建一些容器以便能够把它们放进去，这就是要创建游泳池和热水浴缸给这些人享受的地方。

12.2.4 增加泳池和热水浴缸作为放置目标

就像前面做过的那样，添加一些CSS文件来为HTML提供样式。把如下的CSS文本添加到文档的style标签下去。它们会分别把泳池的背景色设置为蓝色，把热水浴缸的设置为红色：

```
.pool {
    background-color: #CCCCFF;
}
.hotTub {
    background-color: #FFCCCC;
}
```

12

现在需要把HTML代码添加到文档体内。在更衣室HTML代码后面添加如下的HTML标记：

```
<table>
    <tr>
        <td align='center'>
```

```
            Pool
        </td>
        <td align='center'>
            Hot Tub
        </td>
    </tr>
    <tr>
        <td>
            <div id="pool" class="lockerRoom pool"/>
        </td>
        <td>
            <div id="hotTub" class="lockerRoom hotTub"/>
        </td>
    </tr>
</table>
```

这段代码给出了构建放入目标的元素。图12-7展示了现在HTML页面是渲染的样子。

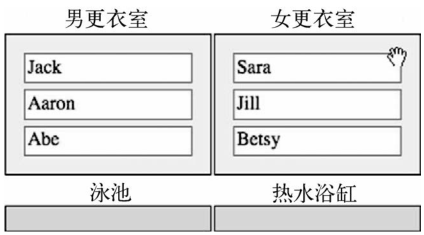


图12-7　渲染在屏幕上的游泳池和热水浴缸

你已经添加了现在所需的所有HTML，现在必须要'pool'和'hotTub'设置为DropTargets了。这使得它们可以作为拖放中的放入部分参与进来。需要添加这段代码到代码清单12-2的Javascript代码之后：

```
var poolDDTarget = new Ext.dd.DDTarget('pool', 'males');
var hotTubDDTarget = new Ext.dd.DDTarget('hotTub', 'females');
```

这里为'pool'和'hotTub'元素分别创建了一个EE.dd.DDTarget的实例。DDTarget构造器的第一个参数是元素的ID（或者是DOM引用）。第二个参数是DDTarget要参与的组。

现在刷新页面并拖放一个男士节点到泳池节点，或者拖放一个女士节点到热水浴缸节点上。当放置节点到目标上时发生什么了？你猜对了：什么都没有发生。为什么呢？对了，这是因为构建了一些拖动元素和放入目标用来完成拖放功能展示，但是记住接下去的开发实现就要你自己负责了。你需要为放入引导、有效或者无效的放入开发代码。你将会看到，这些是最多需要为拖放功能写代码的地方，这也是接下去我们要做的。

12.3　完成你的拖放实现

如前所述，我们很容易设置可拖放的元素，设置为放入目标也是如此。但是，除非你把这些

单独的点元素连接起来，否则结果就是一堆没有路相连的源头和目标。

要添加放入引导、有效及无效的行为，需要重构配置可拖动元素的代码。你将从添加最后的一个CSS类开始，它会把放入目标变为绿色作为放入引导：

```
.dropZoneOver {
    background-color: #99FF99;
}
```

这段CSS很简单，任何有这个类的元素都会有一个绿色的背景。接下来，需要构建一个将作用于每一个`Ext.dd.DD`实例的`overrides`对象，重构一下为男士元素和女士元素添加拖动功能的代码块。

12.3.1　增加放入引导

要增加一个放入引导，你需要完全地替换掉初始化放入目标的代码。下面的代码清单展示了需要用到的东西，它会定义有效和无效的行为。

代码清单12-3　重构`Ext.dd.DD`实现

```
var overrides = {                                              ❶ 创建覆盖对象
    onDragEnter : function(evtObj, targetElId) {         ❷ 添加放入引导
        var targetEl =  Ext.get(targetElId);
        targetEl.addCls('dropZoneOver');
    },
    onDragOut  : function(evtObj, targetElId) {          ❸ 移出放入引导
        var targetEl = Ext.get(targetElId);
        targetEl.toggleCls('dropZoneOver');
    },
    b4StartDrag    : Ext.emptyFn,
    onInvalidDrop : Ext.emptyFn,
    onDragDrop     : Ext.emptyFn,
    endDrag        : Ext.emptyFn
};
var maleElements = Ext.get('maleLockerRoom').select('div');   ❹ 设置男士被拖放元素
Ext.each(maleElements.elements, function(el) {
    var dd = new Ext.dd.DD(el, 'males', {
        isTarget : false                                  ❺ 添加放入逻辑
    });
    Ext.apply(dd, overrides);                             ❻ 为DD实例覆盖方法
});
var femaleElements = Ext.get('femaleLockerRoom').select('div');
Ext.each(femaleElements.elements, function(el) {
    var dd = new Ext.dd.DD(el, 'females', {
        isTarget : false
    });
    Ext.apply(dd, overrides);
});
```

代码清单12-3创建了对象`overrides`，这一对象会被应用到即将会创建的`Ext.DD`实例上。你总共需要覆盖5个方法来获得期望的结果，但是现在仅仅需要覆盖`onDragEnter`❶和

onDragOut❷方法。

记住onDragEnter仅会在拖动元素第一次碰到同组内的拖放元素时被调用。往这一方法中添加了'dropZoneOver'这一CSS类，将会把放入元素的背景颜色改变为绿色，以提供所需要的放入引导。

同样，onDragOut方法仅会在第一次离开同组内的拖放元素时被调用。通过这个方法，引导把放入元素的背景色移除❸。

现在我们先把b4StartDrag、onInvalidDrop、onDragDrop和endDrag这4个方法放到一边，后面再来写它们。我们不想现在就来弄这些，因为想让你把精力集中在添加的这些行为和约束条件上。但如果好奇的话，你会使用b4StartDrag来获得拖动元素的原始*X*、*Y*坐标。这些坐标数据会被用在onInvalidDrop方法上，它会作为临时变量来说明这一方法被触发了。onDragDrop方法会被用来把拖动节点从原始的容器中放入到放入目标容器内。最后，endDrag方法会在invalidDrop属性为true的情形下重设拖动元素的位置。

要使用overrides对象，需要重构初始化包括男士和女士拖动对象❹的代码。这么做是因为想要阻止拖动元素成为一个放入目标，正因此需要在DD构造器上添加第三个参数，这意味着这是一个被限制了一点的对象。你将会明白这里说的被限制了一点点是什么意思。在那个配置参数里把isTarget❺设置为false，这把该拖动元素的控制行为限制成不能成为放入目标。

最后，把overrides对象应用到了新创建的Ext.DD的实例上❻。前面我们说到配置对象仅仅可用来构建有限数量的属性。这么说是因为现在Ext JS中的拖放代码在Ext JS 1.0时代就已经写了，那时大多数构造器直接把配置属性应用到自身上。这就是为什么需要使用Ext.apply来注入覆盖方法，而不是像框架中的大多数构造器那样直接赋值到配置对象上。

你已经增加了引导功能的代码了。让我们来看一下当尝试拖动一个男士节点到泳池或者是热水浴缸时会发生什么吧（参见图12-8）。

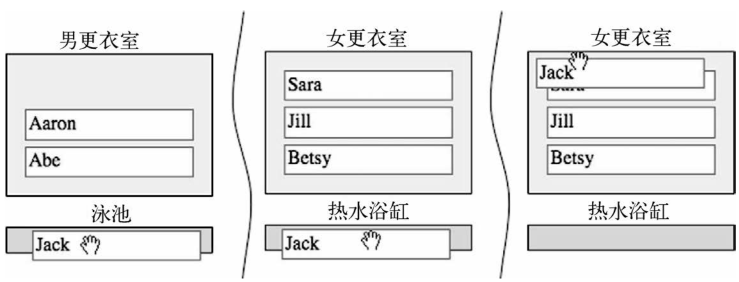


图12-8　男士节点的放入引导条件

基于你对拖放和代码细节的了解，拖动一个男士节点到同组内的放入目标上（想想onDragDrop）将会触发一个放入引导，就是如图所示看到放入目标的背景色变成绿色。当你把一个拖动元素移出同一个放入目标时（想想onDragOut），背景色就会变成原来的状态，放入引导消失了。

相反，拖动一个男士元素到任意其他的放入目标上（比如热水浴缸上）则不会有任何引导提示。为什么会这样呢？'hotTub'元素上没有放入引导是因为热水浴缸和males放入组没有任何关联。

你可能注意到了另外一件事。如图12-9所示，可以拖动一个女士元素到热水浴缸上并触发'hotTub'的放入引导而不是游泳池的。这是因为热水浴缸仅仅和females元素相关联。

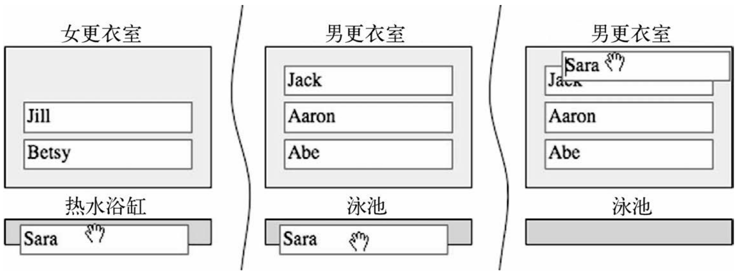


图12-9　女士节点的放入引导条件

虽然这很好地展示了放入引导，但还是应该设置泳池和热水浴缸能够同时接受男士和女士节点；要做到这一点，需要用一个附加组来注册它们。你需要调用addToGroup方法并传入一个替代的组。下面是'pool'和'hotTub'元素注册为DDTarget并附加了addToGroup方法调用：

```
var poolDDTarget = new Ext.dd.DDTarget('pool', 'males');
poolDDTarget.addToGroup('females');
var hotTubDDTarget = new Ext.dd.DDTarget('hotTub', 'females');
hotTubDDTarget.addToGroup('males');
```

在例子中注入这些代码之后，请刷新页面。你将会看到'pool'和'hotTub'放入元素现在引导放入操作了，但是当放入一个拖动元素到有效的放入目标上时，会发生什么？什么都没有发生。这是因为还没有为有效的放入操作写代码。

接下来我们就来完成这一任务。

12.3.2　增加有效放入

要给拖放代码添加有效放入行为，需要替换overrides对象中的onDragDrop方法，如下面的代码清单所示。

代码清单12-4　给overrides添加有效放入

12

```
onDragDrop : function(evtObj, targetElId) {
     var dragEl = Ext.get(this.getEl());
     var dropEl = Ext.get(targetElId);
     if (dragEl.dom.parentNode.id != targetElId) {
         dropEl.appendChild(dragEl);
         this.onDragOut(evtObj, targetElId);
         dragEl.dom.style.position ='';
     }
```

```
        else {
            this.onInvalidDrop();
        }
    }
```

在onDragDrop方法中，写了一次成功（有效）地放入操作的代码。为此，首先需要给拖动和放入元素创建一个本地引用。

下一步有一个if条件语句，它用来判断拖动元素父节点的id是不是和放入目标的id相同。这保证了不会对已经是放入目标的子节点的拖动元素执行放入操作。如果放入目标元素与拖动元素的父节点不同，就允许执行放入操作；否则，调用onInvalidDrop方法，稍后我们就会实现这个方法。

把一个拖动元素从一个父容器移动到另一个父容器的代码是简单的。调用放入元素的appendChild方法，传入放入的元素即可。记住，尽管Ext.dd.DD允许在屏幕上移动拖动元素，它仅仅是改变*X*、*Y*坐标而已。如果不把一个拖动元素移动到另外一个父节点上，它仍然将会属于它原始的容器。

接下来调用了onDragOut覆盖方法，它会消除放入引导。注意，传入了eventObj和targetElId参数给onDragOut方法。正是基于此，onDragOut才能够按照预期完成本职工作。

最后，清除了元素的style.position属性。回想一下，DD设置位置为relative，这在该节点从一个父节点拖放入另一个之后就不需要了。

这样就结束了对onDragDrop方法的覆盖，图12-10展示操作结果。正如图中所显示的，可以成功地把男士和女士元素放到'pool'和'hotTub'这两个放入元素中去，成功地呈现onDragDrop所起的作用。

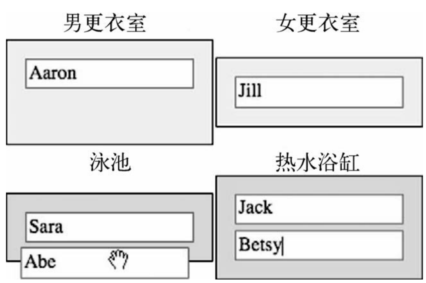


图12-10　男士和女士节点现在可以被放入到游泳池和热水浴缸这两个放入目标上了

虽然说能够把男士和女士元素放入到泳池或者是热水浴缸是一件美事，但是不能让他们永远呆着那里，否则他们会发霉的，你需要能够把他们拉出来并放回到更衣室。尝试把他们拖动回相应的更衣室时又会发生什么呢？没有引导。为什么？对了，因为你没有把更衣室注册成DDTargets。现在就做吧：

```
var mlrDDTarget = new Ext.dd.DDTarget('maleLockerRoom', 'males');
var flrDDTarget = new Ext.dd.DDTarget('femaleLockerRoom', 'females');
```

把这段代码添加到拖放代码的底部，这使得男士拖动元素可以被引导放入到除了女士更衣室的所有放入目标上。同样地，女士拖动元素可以被放入到除了男士更衣室的所有放入目标上。这也符合公共场所很少有男女公用更衣室的惯例。

现在完成了放入操作的所有内容。最后要实现的内容是无效放入行为，这就是你接下来的工作。

12.3.3 实现无效放入

你可能已经注意到，当放入一个节点在屏幕上任意的无效地点，这个节点就会停留在这个无效的地点上。这是因为需要构建无效的放入行为，它会把元素放回到原始的位置上去。你将会用到Ext.fx类作为样式。下面的代码清单替换了overrides对象的b4StartDrag和onInvalidDrop方法。

代码清单12-5 在无效放入之后的清除操作

```
b4StartDrag : function() {                                          ❶ 覆盖b4StartDrag方法
   var dragEl = Ext.get(this.getEl());
   this.originalXY = dragEl.getXY();
},
onInvalidDrop : function() {                                        ❷ 将this.invalidDrop设为true
   this.invalidDrop = true;
},
endDrag : function() {                                              ❸ 动态化拖动元素返回
   if (this.invalidDrop === true) {
      var dragEl = Ext.get(this.getEl());
      var animCfgObj = {
          easing : 'elasticOut',
          duration : 1,
          callback : function() {                                   ❹ 重设拖动元素位置
              dragEl.dom.style.position = '';
          }
      };
      dragEl.moveTo(this.originalXY[0], this.originalXY[1], animCfgObj)   ❺ 动态化重设拖动元素
      delete this.invalidDrop;
   }
}
```

代码清单12-5首先覆盖了b4StartDrag❶方法，这个方法在拖动元素被拖动的瞬间执行。这时可以把拖动元素的原始*X*、*Y*坐标记录下来，用于后续的修复操作。修复一个无效的放入意味着重设拖动元素或者其代理（马上就会提到）的坐标为拖动动作发生前的位置。

接下来覆盖了onInvalidDrop❷方法，它会在拖动元素没有被放入到同组中有效的放入地点的时候被调用。这个方法中所要做的仅仅是把一个本地变量invalidProperty设置成为true，这将会在接下来的方法endDrag中被使用到。

最后覆盖endDrag❸方法，它在本地变量invalidProperty为true时会执行修复操作。它也会用到b4StartDrag设置的本地变量originalXY。这个方法创建了一个配置对象用作动画展示。

在这个配置对象中设置easing为'elasticOut'，这将使得元素在动画结束前有一个平滑的效果，并且把duration设置成一秒钟。这保证了动画效果平滑而不生涩。你还要创建一个回调方法来重设拖动元素的style.position属性❹，这确保了拖动元素完全与它要做的事情符合。

> **注意**　如果放弃动画效果，仅仅是重设一下拖动元素的位置，这样onInvalidDrop只需要把style.position设置为一个空字符串，如：dragEl.dom.style.position = '';。

接下来调用拖动元素的moveTo方法，传入*X*和*Y*坐标值作为第一个和第二个参数，动画效果配置对象作为第三个。这会触发拖动元素的动画效果。

最后需要删除本地引用invalidDrop，因为我们不再需要它了。你需要刷新页面来看这三个覆盖方法是如何工作的。

当拖动一个元素并把它放入到一个不相关联的放入元素上时，它会滑回它原始的位置。在它靠近目标*X*、*Y*坐标位置的时候会有反弹的效果❺。

你现在全面了解如何用Ext.dd.DD和Ext.dd.DDTarget来实现拖放功能了。接下来，我们开发类似的DDProxy类。

12.4　使用 DDProxy

在开发拖放功能过程中，拖动代理是很常见并值得尝试的，因为拖动代理的实现和DD相似但又有所不同。这是因为DDProxy类允许拖动一个拖动元素的轻量级版本，这就是拖动代理。使用DDProxy可以在拖动元素很复杂的情况下极大地提升性能。其中一部分性能的提高来自于DDProxy的每一个实例都在使用DOM组件中同一个代理div。记住拖动代理就是屏幕上被拖动着的元素，这会有助于理解代码实现。

在这个例子中，你还会用到前面已经用的HTML和CSS，如果你打算在实现中使用拖动代理，我们也会提供一个相应的模式。

首先，需要在页面上增加如下一条CSS规则，这将会把拖动代理的背景色设置为黄色：

```
.ddProxy {
    background-color: #FFFF00;
}
```

你将会遵循实现DD类的流程。开发的时候，你会发现实现DDProxy类相较于DD类会多用一些代码。

实现DDProxy类以及放入引导

DDProxy类负责创建和管理可重用的代理元素的*X*、*Y*坐标，并由你负责给它添加样式和内容。你需要覆盖startDrag方法来做到这些，而不是实现DD时用的b4Drag方法。

在代码清单12-6中，将创建一个`DDProxy`实例的`overrides`对象，而要让这段代码清单正常工作，需要用到下面这段CSS代码：

```
.lockerRoom div, .lockerRoomChildren {
    border           : 1px solid #FF0000;
    background-color : #FFFFFF;
    padding          : 2px;
    margin           : 5px;
    cursor           : move;
}
```

这段代码很长，但其实你已经完成了其中不少内容了。

代码清单12-6 实现放入引导

```
var overrides = {
    startDrag : function() {                                    ❶ 覆盖startDrag方法
        var dragProxy = Ext.get(this.getDragEl());
        var dragEl = Ext.get(this.getEl());
        dragProxy.addClass('lockerRoomChildren');               ❷ 样式化DragProxy
        dragProxy.addClass('ddProxy');
        dragProxy.setOpacity(.70);
        dragProxy.update(dragEl.dom.innerHTML);
        dragProxy.setSize(dragEl.getSize())
        this.originalXY = dragEl.getXY();
    },
    onDragEnter : function(evtObj, targetElId) {                ❸ 添加放入引导
        var targetEl = Ext.get(targetElId);
        targetEl.addClass('dropzoneOver');
    },
    onDragOut : function(evtObj, targetElId) {
        var targetEl = Ext.get(targetElId);
        targetEl.removeClass('dropzoneOver');
    },
    onInvalidDrop : function() {
        this.invalidDrop = true;
    },
    onDragDrop : Ext.emptyFn                                    ❹ 添加onDragDrop存根类
};
var maleElements = Ext.get('maleLockerRoom').select('div');
Ext.each(maleElements.elements, function(el) {
    var dd = new Ext.dd.DDProxy(el, 'males', {
        isTarget : false
    });
    Ext.apply(dd, overrides);
});
var femaleElements = Ext.get('femaleLockerRoom').select('div');
Ext.each(femaleElements.elements, function(el) {
    var dd = new Ext.dd.DDProxy(el, 'females', {
        isTarget : false
    });
    Ext.apply(dd, overrides);
});
```

在代码清单12-6中，给代理对象添加了样式、放入引导，并为每个元素初始化了一个Ext.dd.DDProxy实例。下面我们看看它是怎么工作的。

StartDrag方法❶首先给DragProxy元素添加lockerRoomChildren和ddProxy两个CSS类❷，以便给拖动元素提供样式。接着，它把代理对象的不透明度设置为70%，并从拖动元素上把内容复制下来。然后再把DragProxy对象的大小设置为拖动元素的大小。然后还会设置orginalXY属性，这个给接下去可能的无效放入的修复操作使用的。

接下来通过覆盖onDragEnter和onDragDrop方法❸添加放入引导。这和前面的实现完全一样。onInvalidDrop的覆盖也和之前一样。最后要覆盖的是onDragDrop方法❹的存根类，你只需要写一点点代码。

在使用放入引导之前，需要把泳池、热水浴缸和更衣室元素设置为放入目标：

```
var poolDDTarget = new Ext.dd.DDTarget('pool', 'males');
poolDDTarget.addToGroup('females');
var hotTubDDTarget = new Ext.dd.DDTarget('hotTub', 'females');
hotTubDDTarget.addToGroup('males');
var mlrDDTarget = new Ext.dd.DDTarget('maleLockerRoom', 'males');
var flrDDTarget = new Ext.dd.DDTarget('femaleLockerRoom', 'females');
```

现在已经设置好这一切，可以尝试使用一下已经写了一部分的DDProxy了。请刷新页面，然后拖一个拖动元素。图12-11展示了一个拖动代理工作时候的样子。

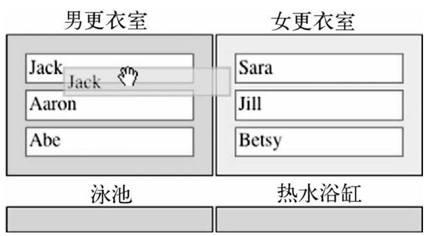


图12-11　DDProxy在一个男士拖动元素上发生作用了

正如你所见到的，在一个可拖动的元素上执行拖动操作会生成出一个DragProxy，它被拖来拖去，但是拖动元素本身却保持原样。你还可以看到放入引导如何工作的。当放入一个拖动元素到一个有效或者无效的放置目标上，会发生什么呢？

在这两种情形下，拖动元素都会被移动到DragProxy的最后可知的坐标地点，这模拟了DD类在没有放入操作有效性约束下的行为。

接下来需要增加这些。下面的代码清单包含了DDProxy的实现。

代码清单12-7　添加有效和无效的放入行为

```
onDragDrop : function(evtObj, targetElId) {          ❶ 覆盖onDragDrop方法
     var dragEl = Ext.get(this.getEl());
     var dropEl = Ext.get(targetElId);
     if (dragEl.dom.parentNode.id != targetElId) {
        dropEl.appendChild(dragEl);
```

```
            this.onDragOut(evtObj, targetElId);
            dragEl.dom.style.position ='';
        }
        else {
            this.onInvalidDrop();
        }
    },
    b4EndDrag : Ext.emptyFn,
    endDrag : function() {
        var dragProxy = Ext.get(this.getDragEl());
        if (this.invalidDrop === true) {
            var dragEl = Ext.get(this.getEl());
            var animCfgObj = {
                easing : 'easeOut',
                duration : .25,
                callback : function() {
                    dragProxy.hide();
                    dragEl.highlight();
                }
            };
            dragProxy.moveTo(this.originalXY[0],
                     this.originalXY[1], animCfgObj);
        }
        else {
            dragProxy.hide();
        }
        delete this.invalidDrop;
    }
```

❷ 禁止代理在拖动结束前隐藏

❸ 覆盖**endDrag**方法

❹ 修复

❺ 如果有效放入，则隐藏手动代理

代码清单12-7完成了剩余的`DDProxy`的实现，增加了`onDragDrop`、`b4EndDrag`和`endDrag`的覆盖方法。

`onDragDrop`❶方法和`DD`实现中是完全一样的，就是当放入元素和拖动元素的父元素不一样时就允许放入操作，并把这个节点移动到放入元素上去。否则就调用`InvalidDrop`方法，这会把`invalidDrop`属性设置为`true`。

`b4EndDrag`方法❷故意用`Ext.emptyFn`（空函数）引用来覆盖。这么做是因为`DDProxy`的`b4EndDrag`方法在`endDrag`方法被调用之前会把`DDProxy`隐藏起来，这和想要执行的动画相冲突了。隐藏`DragProxy`然后又显示出来很浪费，所以用空函数来覆盖`b4EndDrag`方法来阻止它被隐藏。

在先前`DD`的实现中，`endDrag`方法❸用在`invalidDrop`被设置为`true`❹的情形下做修复操作。但是这里需要模拟`DragProxy`，而不是拖动元素本身。通过设置`easeOut`可以使得动画效果更光滑。回调方法会隐藏`DragProxy`，然后调用拖动元素的高亮效果方法，并以动画效果把背景色从黄色变成白色。

最后，如果调用`endDrag`方法的时候，`invalidDrop`属性没有被设置，它就会从视图上隐藏❺代理元素，并完成`DDProxy`的实现。

给通用DOM元素开发全部的拖放功能需要一些工作以及对拖放类层次结构的基本理解，回

报是你可以在屏幕上酷炫地拖放元素，还给用户带来了一些额外的小功能。

让我们继续来看看Ext JS组件上的拖放功能。你可以在我们已经讨论过的基本概念上进行探索。首先，我们来看视图上的拖放。

12.5　视图的拖放

假使要开发一个程序，让经理可以通过简单的拖放手势跟踪雇员在职还是在休假。为此需要构建两个视图，两者和你早先创建过的都很类似。要让它们可用，需要做一些微小的修改，其中包括允许多节点选择。这将是一个在组件上使用拖放的绝佳示例。图12-12展示了在一个`Ext.Window`上封装了两个视图。



图12-12　两个视图

现在知道需要做什么了，让我们开始吧。

12.5.1　构建视图

先要写一些CSS，用来给视图上的元素添加样式。拖放的CSS文件会被包含进来，让我们在下一个代码清单中把它做出来吧。

代码清单12-8　构建视图的CSS代码

```
<style type="text/css">
    .emplWrap {                          ❶ 为整个雇员模板div添加样式
        border: 1px #999999 solid;
        -moz-border-radius: 5px;
        -webkit-border-radius: 5px;
        margin : 3px;
```

```
        padding : 3px;
        background-color: #ffffcc;
    }

    .emplOver {                                    ❷ 设置鼠标悬停样式
        border: 1px #9999ff solid;
        background-color: #ccccff;
        cursor: pointer;
}

    .emplSelected {                                ❸ 设置选中雇员样式
        border: 1px #66ff66 solid;
        background-color: #ccffcc;
        cursor: pointer;
    }

    .emplName {
        font-weight: bold;
        margin-left: 5px;
        font-size: 14px;
        text-decoration: underline;
        color: #333333;
    }

    .emplAddress {
        margin-left: 20px;
    }
</style>
```

在代码清单12-8的CSS中，给视图上的雇员div添加了样式。没有被选择的雇员元素背景色是黄色的❶，和马尼拉文件袋很像。当鼠标悬停在雇员名字上的时候，它将会使用emplOver❷这个CSS类把它变成蓝色。当被选择之后，雇员又会被emplSelected❸这个CSS类渲染成绿色。

现在有了视图将要使用的CSS代码。下面的代码清单会配置两个存储供不同的视图来使用。

代码清单12-9 为数据视图创建存储

```
Ext.define('Employee', {                           ❶ 创建Employee模型
    extend     : 'Ext.data.Model',
    idProperty : 'id',
    fields     : [
        {name : 'departmentName', type : 'string' },
        'departmentName',
        'email',
        { name : 'firstName', mapping : 'firstname' },
        { name : 'lastName', mapping : 'lastname' }
    ]
});
var inOfficeStore = Ext.create('Ext.data.Store', {    ❷ 配置远程存储
    model    : 'Employee',
    autoLoad : true,
    proxy    : {
```

```
        type    : 'jsonp',
        url     : 'http://extjsinaction.com/getEmployees.php',
        reader : {
            type        : 'json',
            root        : 'records',
            idProperty : 'id'
        }
    }
});

var onVacationStore = Ext.create('Ext.data.Store', {      ❸ 创建本地存储
    model: 'Employee'
});
```

在代码清单12-9中，创建一个雇员模型❶和两个存储的配置对象。第一个存储❷用Ajax代理来获取雇员列表，而第二个JsonStore❸则静静地等待放入操作插入进来的记录。

现在数据存储已经配置好，可以创建视图了，如下面的代码清单所示。

代码清单12-10　构造两个视图

```
var dvTpl = new Ext.XTemplate(                            ❶ 为视图提供XTemplate
      '<tpl for=".">',
          '<div class="emplWrap" id="employee_{id}">',
              '<div class="emplName">{lastName}, {firstName}</div>',
              '<div>',
                  '<span class="title">Department:</span>',
                   ' {departmentName}',
              '</div>',
              '<div>',
                  '<span class="title">Email:</span>',
                  '<a href="#">{email}</a>',
              '</div>',
          '</div>',
      '</tpl>'
  );

  var inOfficeDv = Ext.create('Ext.view.View', {          ❷ 创建in-the-office视图
      tpl              : dvTpl,
      store            : inOfficeStore,
      loadingText      : 'loading..',
      multiSelect      : true,
      overItemCls      : 'emplOver',
      selectedItemCls : 'emplSelected',
      itemSelector     : 'div.emplWrap',
      emptyText        : 'No employees in the office.',
      style            : 'overflow:auto; background-color: #FFFFFF;'
  });
  var onVacationDv = Ext.create('Ext.view.View', {        ❸ 创建on-vacation视图
      tpl              : dvTpl,
      store            : onVacationStore,
      loadingText      : 'loading..',
      multiSelect      : true,
      overItemCls      : 'emplOver',
```

```
        selectedItemCls : 'emplSelected',
        itemSelector    : 'div.emplWrap',
        emptyText       : 'No employees on vacation',
        style           : 'overflow:auto; background-color: #FFFFFF;'
    });
```

代码清单12-10在一个通用的XTemplate❶实例的基础上配置和构建了两个视图。inOfficeDv❷会从inOfficeStore中获取数据来加载在在岗的雇员列表，而onVacationDv❸会使用暂时还没有人的onVacationStore。

你可以直接在屏幕上渲染这两个视图。但是，利用HBoxLayout把它们并排地放在一个窗口中会显得更好看，如下面的代码清单所示。

代码清单12-11　把视图放在窗口内

```
new Ext.Window({                                          ❶ 为视图实例化窗口
     layout       : 'hbox',
     height       : 400,
     width        : 550,
     border       : false,
     layoutConfig : { align : 'stretch'},
     items        : [
         {
             title  : 'Employees in the office',
             frame  : true,
             layout : 'fit',
             items  : inOfficeDv,                          ❷ 将视图数据放入面板
             flex   : 1
         },
         {
             title  : 'Employees on vacation',
             frame  : true,
             layout : 'fit',
             id     : "test",
             items  : onVacationDv,
             flex   : 1
         }
     ]
}).show();
```

代码清单12-11创建了一个Ext.Window❶实例，它用了HBoxLayout布局把两个面板并列等宽地放在一起，使它们的高度被拉伸到正好适应窗体。左边的面板是在在岗人员的数据视图❷，而右边的面板是休假中人员的数据视图（参见图12-13）。

视图已经被正确地渲染了，在岗的雇员出现在左边，而且目前没有人正在休假。有了这些之后，你就可以添加拖放操作了。

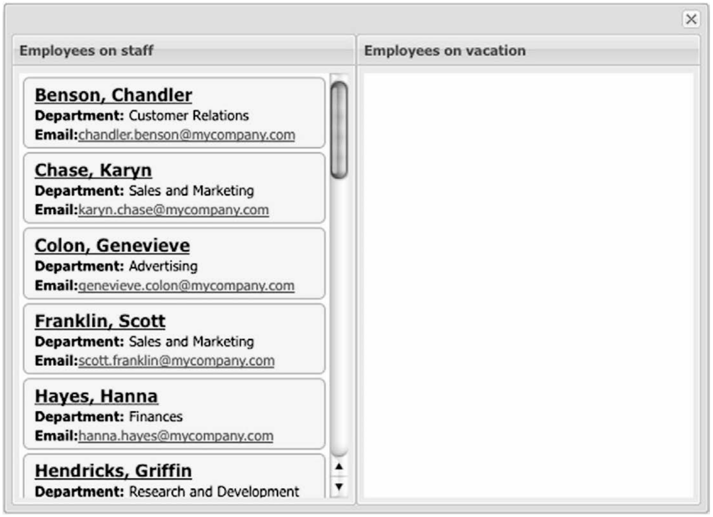


图12-13　`Ext.Window` 组件上渲染的视图

12.5.2　添加拖动手势

视图拖放的应用程序比给网格或者树形面板添加拖放要花多一些力气。这是因为和这些部件不同，`View`类没有自己的`DragZone`子类实现供直接使用，这意味着你不得不制作自己的`DragZone`实现。并且，你还需要开发`DropZone`来管理放入手势。

`DragZone`类使用了一个特殊的叫`StatusProxy`的代理类，它会使用图标来提示这个放入操作是否是成功的。图12-14展示了它们典型情况下的样子。



图12-14　StatusProxy，左边提示可以放入，右边表示不可以

默认的`StatusProxy`是一个非常轻量和高效的，但是有点无趣。虽然它提供了有用的信息，但是用来很是无趣。你应该把`StatusProxy`的定制功能利用起来，把拖动手势做的更花哨一点儿，并使它们更加有趣、信息更丰富。另一个`DragZone`添加的功能是自动修复无效的放入场景，这减轻了你实现这段代码的负担。

首先要为待会儿要写的`DragZone`创建覆盖对象。因为数据视图**必须**要被渲染给拖放使用，你需要在代码清单12-11下面插入如下的代码。

代码清单12-12　创建`DragZone`覆盖

```
var dragZoneOverrides = {
    containerScroll : true,     ← ❶ 滚动目标容器
    scroll          : false,    ← ❷ 禁止document.body滚动
```

```
        getDragData     : function(evtObj){                ❸ 覆盖getDragData方法
            var dataView = this.dataView;
            var sourceEl = evtObj.getTarget(dataView.itemSelector, 10);
            if (sourceEl) {                                 ❹ 缓存拖动手势元素
                var selectedNodes = dataView.getSelectedNodes();
                var dragDropEl = document.createElement('div');

                if (selectedNodes.length < 1) {             ❺ 创建、返回拖动数据对象
                    selectedNodes.push(sourceEl);
                }

                Ext.each(selectedNodes, function(node) {    ❻ 轮询selectedNodes列表
                    dragDropEl.appendChild(node.cloneNode(true));
                });

                return {
                    ddel            : dragDropEl,
                    repairXY        : Ext.fly(sourceEl).getXY(),
                    dragRecords     : dataView.getSelectionModel()
                                        .getSelection(),
                    sourceDataView : dataView
                };
            }
        },
        getRepairXY: function() {
            return this.dragData.repairXY;
        }
    };
```

代码清单12-12创建了覆盖的属性和方法给即将要写的DragZone实例使用。虽然代码总量相对较少，但是有很多逻辑要搞清楚。下面看看它们是如何工作的。

一开始创建两个配置属性，用来在拖动操作过程中控制滚动。第一个是containerScroll❶，它被设置为true。设置这个属性为true之后，DragZone就会调用Ext.dd.ScrollManager.register，来帮助控制DataView的滚动操作。你可以在DragZone实现之后，从DataView上来仔细地观察这一点。

下一个属性，scroll❷被设置成了false。设置成false就阻止document.body在拖动代理被移出浏览器视口的时候滚动。在拖放操作过程中保持浏览器窗口锁定能够提升它操作的有效性。

接下来覆盖了getDragData❸，这对于多节点的拖放程序来说是至关重要的。getDragData的目的是创建一个*拖动数据对象*，你可以看到它会在方法结束时返回了。有一点很重要，getDragData创建并返回的拖动数据对象会被dropZone实例缓存起来，可以通过this.dragData引用来访问它。你可以在后面的getRepairXY方法中看到这一点。

在这个方法中首先要创建一个引用，指向❹拖动手势初始化成sourceEl的那个元素，之后当DataView认为被选中节点的数量不正确时，要用这个引用来更新StatusProxy。你还需要创建一个容器dragDropEl，用来在拖动过程中存放选中节点的副本。这个容器会被放在StatusProxy中。

12

注意　为了接下来方法的流程能继续，需要判断sourceEl是否存在。当注册了DragZone上的元素触发鼠标点下事件时，getDragData就会被调用。这就意味着getDragData甚至会在View元素自身被点击之后被调用，而不单单是其中的一条记录元素，这会导致方法调用失败。

接下来需要查看一下View认为有几个元素在拖动动作中被选中了。如果selectedNodes❺的数量少于1，就把拖动动作开始的那个元素放进去。这么做是因为有时候拖动手势会在View能够注册一个选中的元素之前被初始化。这是对这个奇怪的行为的快速修补。

然后可以使用Ext.each❻方法来轮询selectedNodes链表，并把它加入到dragDropEl上。这会帮助定制StatusProxy，并让用户感觉是在拖动一个或多个选中的节点的副本。

在这段覆盖方法的最后，返回一个用来更新StatusProxy以及任意放置操作的对象。仅仅有一个ddel属性需要被传入这个对象，它会被放到StatusProxy里面。

这段开发中添加了几个其他属性用来定制拖动数据对象。第一个是repairXY，它是存放拖动手势初始化的时候元素的*X*、*Y*坐标的数组。这会在后面的放置修复操作中被用到。

还有dragRecords，它包含了每一个被选择和拖动的元素的Ext.data.Record的实例列表。最后把sourceDataView设置为DataView的引用，以给DragZone使用。dragRecords和sourceDataView两个属性都会帮助DropZone从DataView中删除被放置的记录。

最后一个覆盖列表中的方法是getRepairXY，它会返回一个本地缓存数据对象的repairXY属性，并帮助修复操作在无效放入发生时知道在哪里模拟StatusProxy。

现在已经设置了覆盖方法，是时候实例化一个DragZone实例并把它应用到视图上了，如下面的代码清单所示。

代码清单12-13　在视图上应用DragZone

```
var inOfficeDragZoneCfg = Ext.apply({}, {          ❶ 定制 dragZoneOverrides
     ddGroup      : 'employeeDD',                     的副本
     dataView     : inOfficeDv
 }, dragZoneOverrides);

new Ext.dd.DragZone(inOfficeDv.getEl(), inOfficeDragZoneCfg);

var vacationDragZoneCfg = Ext.apply({}, {
    ddGroup      : 'employeeDD',
    dataView     : onVacationDv
}, dragZoneOverrides);

new Ext.dd.DragZone(onVacationDv.getEl(), vacationDragZoneCfg);
```

在代码清单12-13中，用Ext.apply创建了一个dragZoneOverrides的定制化副本对象作为办公室View❶中定制的DragZone。这个覆盖过的定制的副本会包含ddGroup属性。两个DragZone的实现都会共用这个属性。dataView属性会使得这两个副本特别起来，它会用前面写好的getDragData来引用DragZone附带的DataView。同样的方式可以用在休假DataView的

DragZone上。

你可能注意到了，和DDTarget实现不同，不需要把overrides对象应用到DragZone的实例上。这是因为DragZone的超类DragSource会自动地做这件事，正如Ext.Components那样。

刷新一下页面，现在可以使用拖动操作了。你还能看到定制好的StatusProxy发生作用。如图12-15所示。

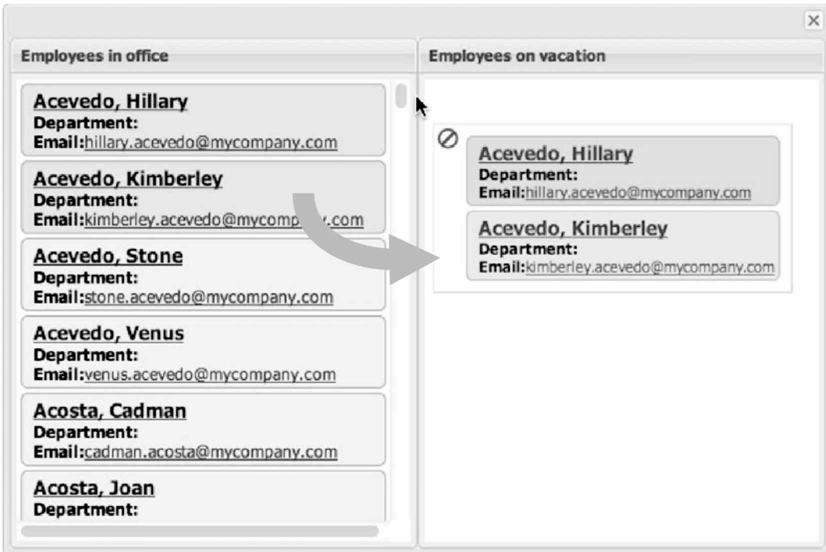


图12-15　带有定制化了的DragProxy的DragZone

可以看到，选择并拖动一个或多个办公室View的Record会把StatusProxy上的选中的节点的副本显现出来，这使得拖动操作更加漂亮，用起来也更有趣。

放置一个拖动代理到页面上任何地方的时候，getRepairXY方法都会起作用。这个动画的效果会把拖动代理滑向拖动操作初始的*X-Y*坐标处。

当试图拖动节点到休假View上时，StatusProxy显示一个图标说明放入操作不能成功。这是因为还没有启用DropZone，这正是我们接下来的任务。

12.5.3　使用放入

正如前面做的拖放应用，这里必须注册一个放入目标来和拖动类交互。前面提到过，你将会用到DropZone类。遵循着这个方式，下面的代码清单创建了一个覆盖对象，它会处理放置手势，而且比拖动手势容易不少。

12

代码清单12-14　创建DropZone覆盖方法

```
var dropZoneOverrides = {
    onContainerOver : function() {
      return this.dropAllowed;
```

❶ 更新StatusProxy

```
        },
        onContainerDrop : function(dropZone, evtObj, dragData) {
            var dragRecords = dragData.dragRecords;
            var store = this.dataView.store;
            var dupFound = false;
            Ext.each(dragRecords, function(record) {          ❷ 搜索重复记录
                var found = store.findBy(function(r) {
                   return r.data.id === record.data.id;
                });
                if (found > -1 ) {
                    dupFound = true;
                }
            });
            if (dupFound !== true) {
                Ext.each(dragRecords, function(record) {        ❸ 移除所有记录
                    dragData.sourceDataView.store.remove(record);
                });
                this.dataView.store.add(dragRecords);          ❹ 添加记录到目标位置
                this.dataView.store.sort('lastname', 'ASC');   ❺ 按姓氏给记录排序
            }
            return true;                                       ❻ 提示放入成功
        }
    };
```

代码清单12-14创建了一个有两个方法的`override`对象，它使得放入手势可以在两个视图中成功发生。第一个方法是`onContainerOver`❶，它用来检测放入操作是否被允许。在这个应用程序中没有什么需要处理的，但是你至少要返回`this.droppedAllowed`引用，这个引用指向了CSS类`x-dd-drop-ok`，这个类提供了一个绿色的核对标记。如果想要一个定制的图标，可以在这里返回一个定制的CSS类。

下一个方法（`onContainerDrop`）是用来处理放入节点的，它会在`mouseup`事件触发的时候，被`DragZone`实例调用。记住`DragZone`是不能和不在同一个拖放组中的`DropZone`交互的。

在这个方法中，会用到覆盖过的`DragZone`上的`getDragData`创建的`dragData`对象。一个选中记录（`dragRecords`） 的本地引用和目标视图的存储（`Store`）被创建用在后续的代码上。

接下来，`onContainerDrop`方法搜寻重复的`Record`❷。如果是在尝试复制而不是移动操作，这很有用。如果没有重复项，`Ext.each`会在`Records`上做循环操作，并把它们从`source-DataView`存储中移除❸。然后这些记录会被添加❹到目标视图的存储上，并且按照姓氏升序排列❺起来。

在所有`Record`操作完成之后，`onContainerDrop`方法会返回一个布尔值`true`。通过返回`true`，告诉`DragZone`放入操作成功了❻，它也就不会触发修复动画了。任何其他的值都意味着放入操作不成功，修复操作会发生。

现在覆盖对象已经有了，是时候把它们应用到视图上了，如下面的代码清单所示。

代码清单12-15　创建`DropZone`覆盖对象

```
var inOfficeDropZoneCfg = Ext.apply({}, {
```

```
    ddGroup           : 'employeeDD',
    dataView          : inOfficeDv
}, dropZoneOverrides);

new Ext.dd.DropZone(inOfficeDv.ownerCt.el, inOfficeDropZoneCfg);

var onVacationDropZoneCfg = Ext.apply({}, {
    ddGroup           : 'employeeDD',
    dataView          : onVacationDv
}, dropZoneOverrides);
new Ext.dd.DropZone(onVacationDv.ownerCt.el, onVacationDropZoneCfg);
```

代码清单12-15给每个视图分别创建了一个`dropZoneOverrides`的定制副本对象，以用作`DropZone`的实现。这和代码清单12-13中创建`DragZone`实例所用的方式是一样的。

现在可以看到，端到端的拖放应用程序可以工作了。刷新页面，并尝试从办公室数据视图上拖动元素到休假数据视图上，如图12-16所示。

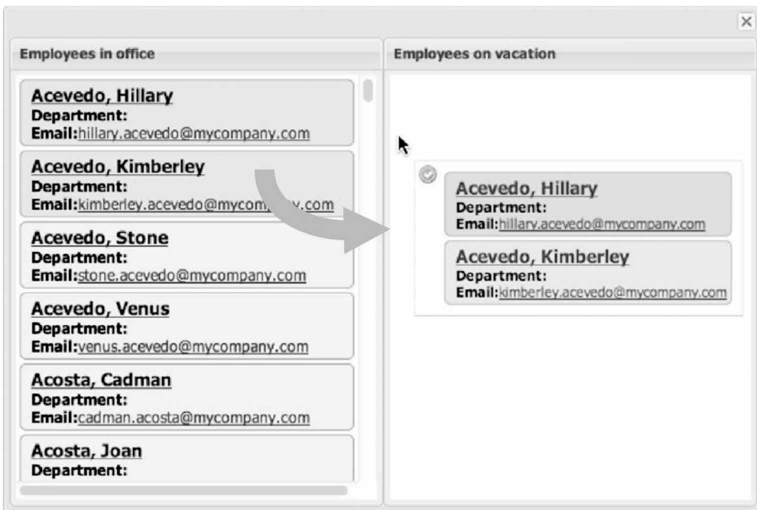


图12-16 `StatusProxy`现在显示可以在放入区域执行放入操作

从雇员视图拖动节点到休假视图上会产生一个`StatusProxy`，它包含了一个绿色的检查标记，来作为放入引导。放入节点的时候会触发`onContainerDrop`方法。如图12-17所示，把`Record`从左边移动到了右边。

就是这样，用一个漂亮的`StatusProxy`来从一个视图拖放元素到另外一个视图。因为每一个视图都有各自附带的`DragZone`和`DropZone`的实例，你可以把元素从一个拖放到另外一个上，`Record`则会被自动地按照名字来排列。

你已经知道如何应用拖放到两个视图上了，并且也知道自己要负责写两种手势的全部端到端的代码。接下来，我们进入网格面板的拖放世界，在那里你会学到一些和视图上拖放不一样的实现方式。这里会使用Ext JS 4提供的拖放插件。

12

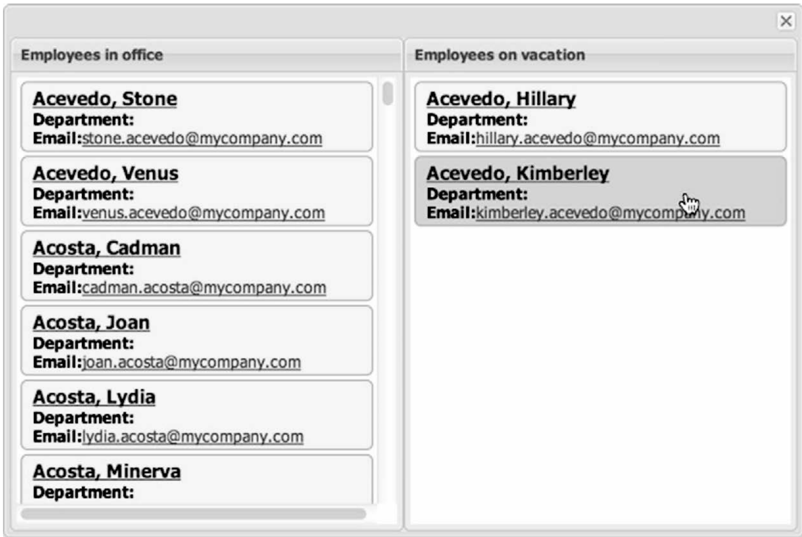


图12-17　你成功地从左边视图拖动了两条`Record`到右边了

12.6　网格面板的拖放

假设要写个程序，让经理用来了解哪些部门需要升级电脑。他们希望能够标记需要升级电脑的部门，并能够调整要升级电脑的部门的顺序。

完成这一工作需要创建两个并排的网格面板，正如前面对视图做过的那样。你需要把拖放应用在网格之间，并允许重排列表中的部门。

在这个练习中，你会发现在网格上实现拖放功能相比在视图上来得容易。你会用到`Ext.grid.plugin.DragDrop`来构建一个特殊的`Ext.dd.DragZone`来和网格面板一起工作。

首先需要构建两个放在窗体中的网格面板。窗体会通过`HBoxLayout`布局来安排网格面板的布局。

构建网格面板

现在你应该能够轻松创建网格面板并配置它们的支持类了。下面的代码清单会创建第一个给这个示例用的网格面板。

代码清单12-16　创建第一个网格面板

```
Ext.define('PCStats', {
    extend: 'Ext.data.Model',
    fields: [
        { name: 'department', type: 'string'},
        { name: 'workstationCount', type: 'int'}
    ]
```

```
});

var remoteJsonStore = {                                          ❶ 创建远程JSON存储
    xtype    : 'json',
    model    : 'PCStats',
    autoLoad : true,
    proxy    : {
        type : 'jsonp',
        url  : 'http://extjsinaction.com/getPCStats.php',
        reader : {
            type : 'json',
            root : 'records'
        }
    }
};
var depsComputersOK = Ext.create('Ext.grid.Panel', {            ❷ 实例化第一个网格面板
    title       : 'Departments with good computers',
    store       : remoteJsonStore,
    multiSelect : true,
    viewConfig  : {
        plugins : {
            ptype : 'gridviewdragdrop'
        }
    },
    columns     : [
        {
            header    : 'Department Name',
            dataIndex : 'department',
            flex      : 1
        },
        {
            header    : '# PCs',
            dataIndex : 'workstationCount',
            width     : 40
        }
    ]
});
```

代码清单12-16用JsonP代理创建了一个远程存储❶。接下来，实例化了一个网格面板❷，它会用存储来呈现各个部门。

接下来的代码清单会创建第二个网格面板，它用来列出需要升级电脑的部门。

代码清单12-17　创建第二个网格面板

```
    var needUpgradeStore = {                                     ❶ 配置本地存储
        xtype : 'json',
        model : 'PCStats'
    };

    var needUpgradeGrid = Ext.create('Ext.grid.GridPanel', {    ❷ 配置第二个网格面板
        title             : 'Departments that need upgrades',
        store             : needUpgradeStore,
        multiSelect     : true,
```

```
        viewConfig       : {
            plugins: {
                ptype: 'gridviewdragdrop'        <-- ❸ 配置gridviewdragdrop插件
            }
        },
        columns          : [
            {
                header    : 'Department Name',
                dataIndex : 'department',
                flex      : 1
            },
            {
                header    : '# PCs',
                dataIndex : 'workstationCount',
                width     : 40
            }
        ]
    });
```

代码清单12-17用`PCStats`模型配置了一个本地存储❶，然后给需要升级的部门创建了第二个网格面板❷，最后配置了gridviewdragdrop插件❸。

这些网格面板需要一个“家”。下面的代码清单展示了如何创建一个用来放置它们的`Ext.Window`。

代码清单12-18　给网格面板创建一个“家”

```
new Ext.Window({
    width    : 500,
    height   : 300,
    border   : false,
    defaults : {
        frame : true,
        flex  : 1
    },
    layout : {
        type : 'hbox',
        align : 'stretch'
    },
    items        : [
        depsComputersOK,
        needUpgradeGrid
    ]
}).show();
```

代码清单12-18创建了一个`Ext.Window`，它用`HBoxLayout`布局来管理两个网格面板。是时候把面板拿出来测试一下了（结果如图12-18所示）。

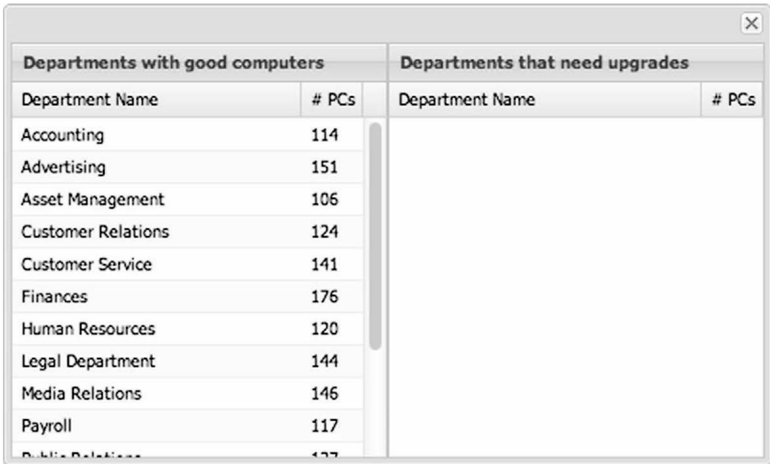


图12-18　两个并排的部门网格面板

如果从左边的网格中选择一个元素并拖放到右边的网格中，你会看到如图12-19所示的结果。

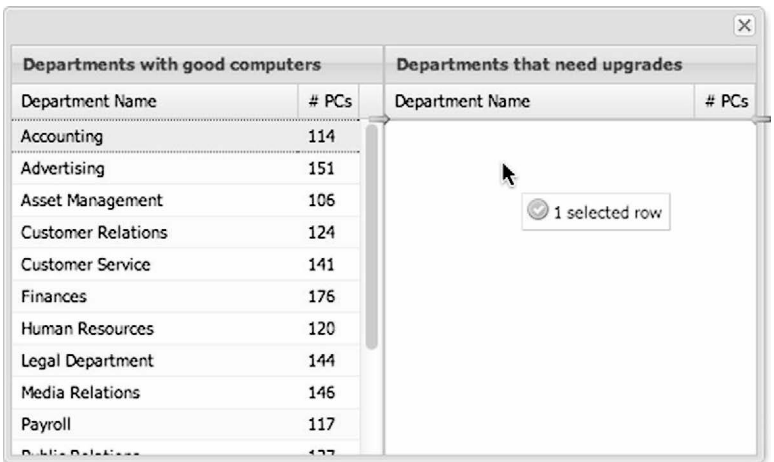


图12-19　网格面板上允许拖动手势

当在左边的网格面板上尝试拖动手势的时候，可以看到`StatusProxy`会显示被选中的行数。这就是`GridDragZone`类使用`getDragData`的方式，它会显示拖动数据对象的`ddel`属性的数量。听上去似曾相识？这里用`gridviewDragdrop`插件走了一个捷径，如果深入研究一下代码，你会发现它用了和前面一样的框架来拖放操作。这很简单，不是吗？其实树形面板也有同样的插件。你可以以同样的方式来使用它，但是让我们不用插件来研究下如何在树形面板上实现拖放功能。

12.7　树形面板上的拖放

现在，公司收购了另外一家公司，管理层需要一个方式来跟踪如何从收购的公司的多个不同部门吸收雇员。他们要你来开发这样一个程序，让他们可以用树形面板和拖放来跟踪雇员的重新分配。

这里最终的需求是能够允许把同事重新安排到一个相似的部门的特殊组织下。比如说，任何来自审计、财务或者薪资管理部门的同事可以被重新安排到任意这些部门之中。同理，来自客户关系、媒体关系、客户服务和公共关系部门的同事则可以被重新安排到这其中的任意部门。不像在JavaScript中那样需要创建一个有效放置矩阵，这里每一个元素都会从服务器上拿一个返回的节点列表作为有效部门。你需要选择一个可以满足需求的数据。

为了大致了解如何使用这些约束，我们一起看下给树形面板使用的雇员记录：

```
{
    "text": "Kemp, Sawyer",
    "leaf": true,
    "validDropPoints": [
        "Accounting",
        "Finances",
        "Payroll"
    ]
}
```

深入研究放入操作的实现逻辑时，会使用`validDropPoints`数组来驱动图形界面的决策过程。对于这条记录，这个雇员仅可以被移动到审计、财务或者薪资管理部门。

好了，让我们开始构建一个树形面板吧。之后，需要添加一些拖放功能。

12.7.1　构建树形面板

正如前面视图和网格面板的例子一样，这里需要构建两个树形面板，它们都会被一个`Ext.Window`的实例用`HBoxLayout`布局管理起来。

因为已经写过几个树形面板了，我们打算在这里只做简单介绍，如下面的代码清单所示。

代码清单12-19　为树形面板上的拖放做准备

```
var leftTree = Ext.create('Ext.tree.Panel', {          ❶ 他们公司的面板
    autoScroll : true,
    title      : 'Their Company',
    animate    : false,
    store      : Ext.create('Ext.data.TreeStore', {
        proxy : {
            type   : 'jsonp',
            url    : 'http://extjsinaction.com/theirCompany.php',
            reader : {
                root : 'records'
            }
        },
```

```
            Root  : {
                text     : 'Their Company',
                id       : 'theirCompany',
                expanded : true
            }
        })
    });

    var rightTree = Ext.create('Ext.tree.Panel', {          ❷ 你公司的面板
        title      : 'Our Company',
        autoScroll : true,
        animate    : false,
        store      : Ext.create('Ext.data.TreeStore', {
            proxy : {
                type   : 'jsonp',
                url    : 'http://extjsinaction.com/ourCompany.php',
                reader : {
                    root : 'records'
                }
            },
            root  : {
                text     : 'Our Company',
                expanded : true
            }
        })
    });

    // 拖放代码

    Ext.create('Ext.Window', {                               ❸ 包含树形面板的窗口
        Height   : 350,
        width    : 450,
        border   : false,
        layout   : {
            layout : 'hbox',
            align  : 'stretch'
        },
        defaults : {
            flex : 1
        },
        items    : [
            leftTree,
            rightTree
        ]
    }).show();
```

在代码清单12-19中，创建了两个视图面板和一个用来容纳并用`HBoxLayout`布局它们的`Ext.Window`。左边的树形面板❶会加载其他公司的部门列表。每一个部门都要能够被展开，以显示子节点。

右边的树形面板❷则会加载本公司的雇员列表，幸运的是，它们和被卖的公司部门是一致的。简单起见，我们不想展示公司现有的雇员了。

12

最后，创建Ext.Window❸来管理两个并列的树形面板。图12-20展示了树形面板在屏幕上的样子。

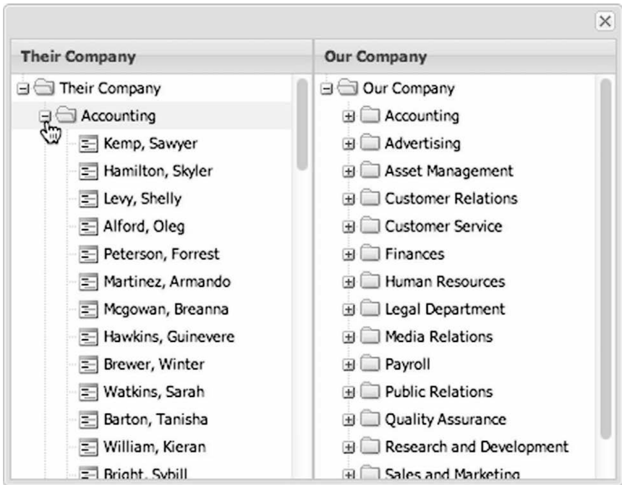


图12-20　两个树形面板

现在已经有了两个渲染在Ext.Window上的树形面板，是时候来关注一下拖放了。

12.7.2　启用拖放

当探索如何在数据视图上使用拖放时，需要实现DragZone和DropZone两个类。当特性在网格面板上实现的时候仅仅需要实现DropZone，因为如果网格面板设置了enableDragDrop属性的时候网格视图已经自动地创建了GridDragZone。

对树形面板来说，可以通过启用ViewDragDrop插件轻松地获得拖放功能。通过添加如下的配置属性到树形面板上即可：

```
viewConfig : {
    plugins : { ptype : 'treeviewdragdrop' }
}
```

通过这种方式启用拖放功能，树形面板上的拖放操作变简单了。每一样东西都可以被拖放，这对于文件系统管理工具等非常实用，但是对于你的需求，基本的拖放实现就无法胜任了。给现成的拖放插件添加约束变得非常困难。

为了能够应用约束，你将要实现自己的ViewDragZone和ViewDropZone类的实例。接下来我们解决这一难题，准备好了吗？

12.7.3　采用灵活的约束

把解决这一问题分成两个阶段：启用在左边树形面板的拖动操作，然后启用右边树形面板的

放入操作。在左边的树形面板上启用拖动操作是两者中最简单的，接下来的代码清单中会展示这一点。下面的代码会被放到在代码清单12-19中你创建Window实例之前。

代码清单12-20 应用更好的放入约束

```
leftTree.getView().on('render', function(view) {          ◁ ❶ 添加渲染监听器
  Ext.create('Ext.tree.ViewDragZone', {          ◁ ❷ 创建ViewDragZone
       view         : view,
       dragText     : 'schedule vacation',
        ddGroup      : 'myTreeDDGroup',
        onBeforeDrag : function(dragData) {
            return view.getNode(dragData.item).attributes.leaf;          ◁ ❸ 仅在叶子节点上启用拖动功能
        }
     });
});
```

为了启用拖动，需要在左边的树形视图类上注册一个渲染监听器❶。这个监听器负责创建一个ViewDragZone的实例❷。这个ViewDragZone类需要一些基本的非常明了的配置参数，我想在这儿提一下onBeforeDrag函数❸。

在此方式下，你需要实现自己的onBeforeDrag函数，如果被拖动到元素有叶属性为true，则函数必须返回为true。这样的效果是拖动操作仅可以被应用在叶子节点（雇员）上，而不能是分支节点（部门）上。

拖动操作完成并绑定之后便可以在右边的树形面板上实现放入操作了。需要把如下的代码清单放到代码清单12-20的代码之后。

代码清单12-21 应用更好的放入约束

```
添加渲染监听器 ❶ ▷ rightTree.getView().on('render', function(view) {
  Ext.create('Ext.tree.ViewDropZone', {          ◁ ❷ 创建ViewDropZone
     view             : view,
     ddGroup          : 'myTreeDDGroup',
     isValidDropPoint : function(node, pos, dz, e, data) {          ◁ ❸ 覆盖isValidDropPoint

         var dropNode   = view.getRecord(data.item),
             targetNode = view.getRecord(node),
             dragNode   = data.records[0],
             validDropPoints,
             targetNodeText;

         if (! dropNode || !targetNode) {
             return false;
         }

         if (targetNode.raw.leaf) {
             return false;
         }

         if (pos != 'append') {
             return false;
         }
```

```
            validDropPoints = dragNode.raw.validDropPoints;
            targetNodeText = targetNode.get("text");

            return Ext.Array.contains(validDropPoints, targetNodeText);
        }
    });
});
```

添加一次验证测试 ❹

代码清单12-21包含了确保放入操作仅仅会发生在相关联的分支节点上的所有代码。为了能够在正确的时间创建ViewDropZone，需要在右边的树形View类上注册一个渲染事件监听器❶。

这个事件监听器唯一的功能是创建ViewDropZone类的实例❷。为了添加合适的放置约束，需要覆盖isValidDropPoint方法❸。

在这个方法中，你从前面的词汇范围引用中收集一些数据。然后会有一些if判断，如果在遇到满足约束条件的情形下就会返回false；如果没有约束被满足，就会返回targetNodeText是否包含在validDropPoints数组中的验证结果❹。其中的约束包括试图放入到：

- 非树形视图上的节点；
- 树形视图上叶子节点；
- 没有给位置（pos）append属性的节点。

假设放置操作通过了所有这些验证，然后返回了调用Ext.Array.contains方法的结果，你可以通过它看到targetNode（放入目标）的名称是否匹配放入节点（拖动节点）validDropPoints数组。如果匹配，就允许放置，如图12-21所示。

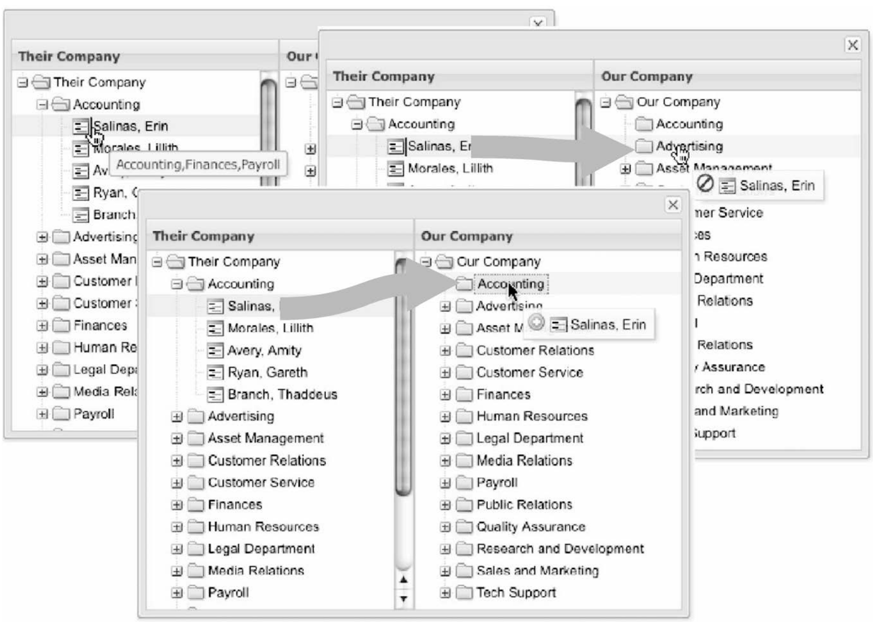


图12-21　检测放入目标约束的逻辑

你可以通过尝试拖动一个部门分支节点来检验约束逻辑，并可以发现那是不可能的，这表明onBeforeDrag覆盖方法如设计的那样工作了。

要决定节点可以被拖动到哪儿，把鼠标悬停上去，就可以看到ToolTip提示框，里面有从validDropPoints数组中定义的值。当鼠标悬停到Erin上的时候，你会发现她可以被放入到会计部门而不能是广告部。当把Erin拖动到右边树形面板上的会计部门时，StatusProxy会展示一个有效的放入图标。但是如果是把她停在广告部门上，你就会看到StatusProxy显示一个无效的放入图标。此外，如果尝试把Erin停到财务部门和薪资管理部门上，你同样会看到一个有效的放入图标。

最后一个验证的是把一个或两个同事放置到有效的部门上。你还可以测试是否可以放入叶子节点到一个有效部门的其他叶子节点的上面或者下面。

就是这些了：两个树形面板之间的拖放，放置约束系统有点复杂但是很灵活。我相信你的工作表现足以让经理满意了。

12.8　小结

本章研究了在三种框架中最常用的部件上实现拖放的多种方式。首先是在DOM元素上实现拖放。本章涵盖了拖放框架的基础知识，然后讲了一个较复杂的基于视图的例子。然后，你探究了实现拖放的捷径：如何在网格面板上使用插件。最后讲解了如何在树形面板上实现拖放。在这个实现中可以看到，在树形面板上启用拖放是很简单的任务，但是放置手势的约束条件确实非常复杂。

下一章将介绍插件和扩展及其工作原理。你将通过JavaScript使用面向对象技术，并且获得很多乐趣。

Part 3 第三部分

构建一个应用

在这一部分，你将了解Ext JS的工作方式，并掌握一些构建MVC应用的最佳实践。

第13章将关注原型继承、类定义、子类化组件、框架扩展、通用插件、动态类加载器（包含复杂的带有上下文菜单的网格面板，而上下文菜单可用于各种数据视图），借此深入探讨用于加强重用性的类系统基础知识。第14章展示了如何构建基于MVC架构和控制器的复杂应用，Ext JS、JavaScript和CSS的代码规范，以及推荐的运用Sencha的命令行工具的开发过程，这涉及了贯穿本书的知识点。

这一部分使你能够使用框架的更高级功能（比如定制的扩展、插件，以及类加载器），并且还会学到构建和管理Web应用的坚实理论。

第 13 章 类系统基础

本章内容

- ❑ 理解原型继承
- ❑ 开发第一个扩展
- ❑ 使用插件
- ❑ 探究Ext JS的类加载器

每个Ext JS开发人员都会面对很多挑战，其中之一便是可重用性。一般来说，一个组件在应用的生命周期中都会多次出现。如果不掌握一些必要的技能，最终的代码可能会变成“函数汤”——难以维护。因此，我们需要关注如何使用框架提供的扩展和插件来实现可重用性。

本章将介绍Ext JS基本的扩展方法（子类化）。首先，你将学习在JavaScript中创建子类，然后了解如何通过原生语言工具来完成任务。这一知识可为重构新写的子类以使用Ext JS类系统打下基础。

一旦你能熟练地创建基本的子类，我们就来学习扩展Ext JS的组件。你会学习框架扩展的基础知识，然后通过扩展网格面板部件来解决一个现实问题。

接下来你会看到，若通过扩展来解决问题，当多个部件需要用到类似的功能时会产生继承问题。一旦你理解了扩展的基本限制，就会想把扩展转换成插件，这样它的功能就容易被网格面板以及其子类共用了。

当你扎实地掌握了Ext JS类系统的基础知识之后，我们一起看下Ext JS提供的动态类加载器。我们会探讨使用动态类加载器的三种流行模式以及相应的注意事项。注意，我们建议大家在学习过程中下载本章相应的示例代码，并在阅读时随时查看。

13.1 经典的 JavaScript 继承

JavaScript提供了原型继承的所有可能的工具，但是构建多重类继承就显得不足了。Ext JS通过类系统让多重类继承变得容易不少。开始学习继承前，你先要创建一个基类。

为了帮助学习接下来的内容，我们假设你正在做汽车销售，负责出售两种汽车。第一种是基础车型，它是高价车型的构建基础。你将使用JavaScript类来描述这两种汽车模型，而不是3D模型。

注意 如果是第一次接触面向对象JavaScript亦或感觉有点生疏，Mozilla基金会有一篇极好的文章来提升强化这方面的技能。你可以访问http://mng.bz/R9BB找到它。

13.1.1 创建一个基类

首先构建一个描述基础车型的类，如下面的代码清单所示。

代码清单13-1 构建一个基类

```
var BaseCar = function(config) {              ← ❶ 创建构造器
    this.octaneRequired = 86;
    this.shiftTo = function(gear) {
        this.gear = gear;
    };
    this.shiftTo('park');
};
BaseCar.prototype = {                         ← ❷ 指派原型对象
    engine    : 'I4',
    turbo     : false,
    wheels    : 'basic',
    getEngine : function() {
        return this.engine;
    },
    drive     : function() {
        console.log("Vrrrroooooooom - I'm driving!");
    }
};
```

代码清单13-1创建了`BaseCar`的构造器❶，当实例化的时候，它会给实例的本地属性`this.octaneRequired`赋值，添加一个`this.shiftTo`方法，并调用它，把本地属性`this.gear`赋值为`'park'`。然后配置`BaseCar`类的`prototype`对象❷，它包含了三个描述`BaseCar`的属性，以及两个方法。

使用如下的代码创建`BaseCar`的一个实例，并用Firebug来检查它的内容：

```
var mySlowCar = new BaseCar();
mySlowCar.drive();
console.log(mySlowCar.getEngine());
console.log('mySlowCar contents:');
console.dir(mySlowCar)
```

图13-1展示了在Firebug多行编辑器和控制台上的代码输出结果。

现在可以关注`BaseCar`类的子类化了。首先以传统的方式来进行。这样一来，稍后使用`Ext.define`时能够更好地理解底层发生的事情。

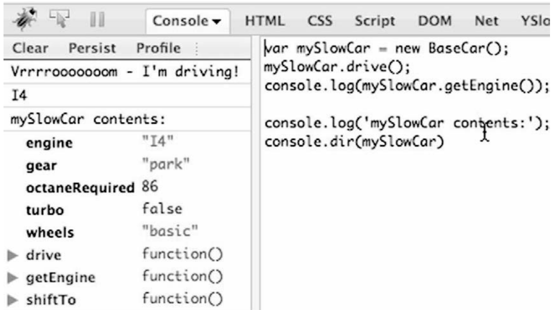


图13-1　实例化基础车型的实例并操作了它的两种方法

13.1.2　创建一个子类

你几步就能用原生的JavaScript来创建一个子类。我们并非仅仅描述每一步怎么做，而是会一步步地介绍实际操作。下面的代码清单展示了如何创建`BaseCar`的子类`PremiumCar`。

代码清单13-2　用老派的方式创建一个子类

```
var PremiumCar = function() {
    PremiumCar.superclass.constructor.call(this);
    this.octaneRequired = 93;
};
PremiumCar.prototype   = new BaseCar();
PremiumCar.superclass  = BaseCar.prototype;
PremiumCar.prototype.turbo  = true;
PremiumCar.prototype.wheels = 'premium';
PremiumCar.prototype.drive = function() {
    this.shiftTo('drive');
    PremiumCar.superclass.drive.call(this);
};
PremiumCar.prototype.getEngine = function() {
    return 'Turbo ' + this.engine;
};
```

❶ 配置子类构造器

❷ 调用超类构造器

❸ 设置子类原型

❹ 设置子类的超类引用

要创建一个子类，首先要创建一个新的构造器，并把它分配给引用`PremiumCar`❶。这个构造器内会调用`Premium.superClass`的`constructor`方法。其作用域是用当前正在创建的`PremiumCar`❷（`this`）的。

之所以要这么做，是因为JavaScript与其他的面向对象语言不同，它的子类不会原生地调用它们超类的构造器。调用超类的构造器会为其带来一个机会，即实施和完成子类可能需要的任何基于构造器的特定功能。在这个例子中，`shiftTo`方法是由`BaseCar`构造器添加并调用的。如果不调用超类的构造器，那就意味着子类不能获得基类构造器提供的好处了。

然后，把`PremiumCar`的`prototype`属性设置为`BaseCar`❸的一个新的实例。这一步操作使得`PremiumCar.prototype`能够继承`BaseCar`所有的属性和方法。这称作为*原型继承*，它是JavaScript中构建类层次结构最为通用和健壮的方式。

接下来的一行代码把`PremiumCar`的`superclass`引用设置为`BaseCar`类❹的`prototype`值。然后你就可以用这个`superclass`的引用去做一些事情了，其中有创建扩展方法，比如`PremiumCar.prototype.drive`。这个方法称作扩展方法，因为它调用超类的原型上类似名字的方法， 但是是从它的子类实例的作用域上调用的。

提示 所有的JavaScript函数（JavaScript 1.3及之后版本）都有两个方法强制作用域执行：`call`和`apply`。要学习`call`和`apply`，请访问www.webreference.com/js/column26/apply.html。

有了子类之后就可以通过如下代码实例化一个`PremiumCar`的实例，并进入Firebug编辑器中进行测试了：

```
var myFastCar = new PremiumCar();
myFastCar.drive();
console.log('myFastCar contents:');
console.dir(myFastCar);
```

图13-2展示了Firebug中的输出。这些输出表明子类做了你预期的事情。从`console.dir`输出中，可以看到子类构造器把`octaneRequired`属性设置为`93`，并且`drive`扩展方法甚至把`gear`方法设置为`"drive"`。

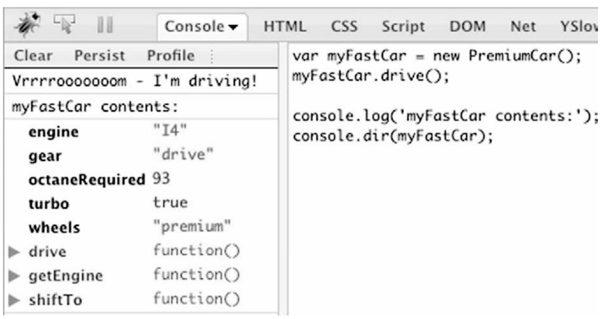


图13-2 运行中的`PremiumCar`子类

这个练习表明了你需要对使用原生JavaScript语言实现原型继承的所有关键步骤负责。首先需要创建子类的构造器，然后还需要把子类的原型设置成基类的实例，再然后要为了方便设置一下子类的`superclass`引用。最后，你需要一个个地给原型添加成员变量。

可以看到，通过原生语言结构创建多个类需要遵循很多步骤。幸运的是，实现相同的结果有更简单的办法。接下来，你将看到Ext JS类系统是如何简化类创建和多重继承的。

13.2 Ext JS 的继承

Ext JS的类系统把JavaScript的原型继承带到了更高的层次上，增加了不少功能，比如依赖注入、自动创建获取方法和设置方法、静态以及掺入支持（多重继承）。所有这些功能都需要用一

些稍后将学到的Ext JS类方法，比如Ext.define、Ext.create以及Ext.require。通读本节内容，你就会知道为什么Ext JS的类系统是一个很棒的应用开发解决方案了。

13.2.1　创建一个基类

你已经看到如何实现JavaScript原型继承了，知道要做到简单级别的继承就不得不做相当多的工作。当碰到像这些应用一样的复杂软件的时候，你不得不敲很多的代码才能实现继承。这意味着项目中会有冗余代码，并导致代码膨胀。

Ext JS是一个完美的选择，它可以替你做这些繁重的工作。要想知道为什么我们这么说，请回顾最初创建的两个类。代码清单13-3展示了开始用Ext JS创建BaseCar类时BaseCar和PremiumCar类的样子。然后通过扩展BaseCar来创建PremiumCar类。这里通过合适的带有命名空间的包名来定义类名，而不是使用简单的类名，这一点和一些经典编程语言很像，比如Java。

代码清单13-3　使用Ext JS定义基类

```
Ext.define('MyApp.car.BaseCar', {                    ❶ 定义基本车型类（BaseCar）
    engine : 'I4',
    turbo : false,
    wheels : 'basic',                                 ❷ 指定类的基本类型

    constructor : function(config) {
        this.octaneRequired = 86;                     ❸ 定义构造器

        this.shiftTo = function(gear) {
            this.gear = gear;
        };

        this.shiftTo('park');
    },
    getEngine : function() {
        return this.engine;
    },
    drive     : function() {
        console.log("Vrrrroooooom - I'm driving!");
    }
});
```

代码清单13-3呈现了Ext.define最基本的使用方式，用它定义了BaseCar类❶。第一件怪事是定义类名的方式：通过字符串定义。这是因为Ext JS给了你把类名和命名空间定义在一起的机会。如果先前没有创建过，Ext JS会在命名空间MyApp.car创建它，并且还会把BaseCar类放到那个命名空间下。这一模式给我们这章中将要学习的类加载系统铺好了道路。在那儿你会知道根据命名空间在文件系统中组织类文件是多么重要。

在定义类的时候，给类的原型❷的基本类型赋值，然后创建构造器❸。这个类和在代码清单13-1定义的类有一模一样的行为，不同的是在用Ext JS的方式定义它。

要实例化这样的一个类，需要用Ext.create而不是JavaScript的关键字new。现在你已经习惯了使用Ext.create来使用Ext JS的类了，但是还要学会把它用在自己定义的类上。可以这么做：

```
var mySlowCar = Ext.create('MyApp.car.BaseCar');
mySlowCar.drive();
console.log(mySlowCar.getEngine());
```

图13-3展示了结果。

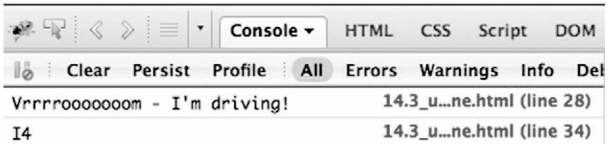


图13-3　第一个Ext JS 类的结果

如图13-3所示，你从刚刚实现的MyApp.car.BaseCar类得到了预期的结果。这为通过Ext.define来扩展这个类创造了完美的条件。

13.2.2　创建一个子类

下面这个代码清单展示了如何扩展BaseCar：使用Ext.define方法来创建超类MyApp.car.BaseCar的一个扩展（也就是子类）：Myapp.car.PremiumCar。下面介绍它是怎么工作的。

代码清单13-4　使用Ext.define扩展BaseCar

```
Ext.define('MyApp.car.PremiumCar', {
    extend      : 'MyApp.car.BaseCar',

    turbo       : true,
    wheels      : 'premium',
    stereo      : '5.1',

    constructor : function() {
        this.callParent(arguments);
        this.octaneRequired = 93;
    },
    getEngine   : function() {
        return 'Turbo ' + this.engine;
    },

    drive       : function() {
        this.callParent();
        this.shiftTo('drive');
        console.log('The turbo makes a big difference!');
    }
});
```

❶ 定义高级车型类
❷ 扩展基本车型类
❸ 覆盖基本车型原始基本类型
❹ 定义高级车型构造器
❺ 超类的扩展驱动方法

首先调用Ext.define来定义一个Myapp.car.PremiumCar❶扩展类，把先前定义的MyApp.car.BaseCar作为一个字符串赋值给extend关键字❷来告诉Ext JS要扩展这个类。接下来，设置原型覆盖，把这个类的turbo、wheels和stereo属性设置得和父类不同。

13

在考虑扩展类的时候，必须要考虑子类中的原型方法是不是要和父类中的原型方法共用方法名。它们是否使用同样的符号引用，将决定它们是扩展方法还是覆盖方法。

扩展方法是子类的方法，它和父类中的方法拥有同样的名称。之所以是扩展方法，这是因为事实上这一方法内部会执行父类中的这一方法。使用扩展方法的原因是它可以减少重复代码，这样就可以重用父类中的代码了。

这个类的构造器❸方法是一个扩展方法。这和前面创建的`PremiumCar`构造器❹是完全一样的，需要加上`this.callParent`（`arguments`）来调用父类的构造器❺方法。这个语句让子类可以获得构造器方法的调用链，有效地让超类的构造器`MyApp.car.BaseCar`能够在子类`MyApp.car.PremiumCar`实例的作用域上来执行。

*覆盖方法*是子类中的方法，它和基类中的另一个方法共用同样的引用名称，但是它不会用`this.callParent()`来链式地调用超类中的方法。如果希望舍弃基类中同名的代码，可以覆盖一个方法。因此，不需要在覆盖方法内调用`this.callParent()`方法。

现在已经用`Ext.define`配置了`PremiumCar`类，可以用Firebug来看看效果了。可以重用测试手动创建的子类的代码来做测试。

```
var myFastCar = Ext.create('MyApp.car.PremiumCar');
myFastCar.drive();
console.log(myFastCar.getEngine());
```

图13-4展现了输出在Firebug控制台的样子。

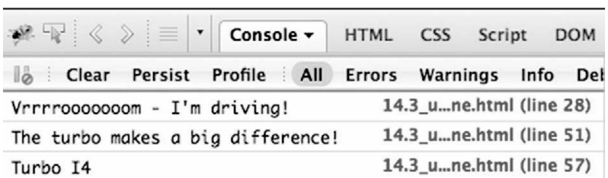


图13-4　实例化`PremiumCar`类的结果

刚刚已经成功通过`Ext.define`扩展了一个类：首先构建一个类（`MyApp.car.BaseCar`），然后扩展它（`MyApp.car.PremiumCar`）。你可以扩展`MyApp.car.PremiumCar`，并创建`MyApp.car.SportsCar`类以添加一些功能（比如`drift()`或者`dragRace()`）。

现在，你已经知道如何用JavaScript和`Ext.define`做原型继承了，可以在Ext JS中使用扩展了。

13.3　扩展 Ext JS 组件

开发框架的扩展是为了在重用已经存在的类的基础上引入新的功能。重用性驱动了框架，使用得当的话，它可以提升应用的开发。

一些开发人员创建一些预先配置好的类，它通过把配置参数直接填入到类的定义上来减少应

用级别的代码。这样的扩展可以减轻应用代码管理大量的配置，它们只需要简单地实例化这些类就可以了。这种设计模式是不错的，但是应该只有在期望杜绝一个预配置的类有多个实例的情况下使用。如果定义一个类仅仅把它用来放一系列的配置参数是很浪费的。

其他的扩展添加一些功能，譬如工具方法，或者类内部嵌入的行为逻辑。一个这样的例子是表单面板在保存失败的时候自动地弹出一个消息框。在部件包含一些受限的内置行为逻辑的地方，Jesus经常因为这个特殊的原因为应用程序创建一些扩展。之所以说受限，是因为Ext JS 4.0现在已经有了MVC架构，它允许把业务逻辑抽象到控制层上。这是我们下一章将要讨论的。

我最喜欢的扩展是一种我称为组合部件的东西，它能把一个或多个部件合并到一个类中。举两个例子：有一个内置的网格面板的一个窗口，或者一个内置了标签面板以在多个标签中分布字段的表单面板。

这就是我们即将要关注的扩展类型。你将要把网格面板和菜单合并起来。

13.3.1 想想你在构建什么

当构建扩展的时候，你可能想要退一步来分析一下来自所有方面的问题。有时候问题可能非常复杂，比如创建一个魔术般的动态部件，它包含大量的必须被UI控制的工作流规则。通常，扩展可以用来解决重用的问题。这将是我们在剩下来的章节中要重点来看的。

回想一下，我们在第8章中研究了网格面板的创建。你附加了一个菜单并在网格的`contextmenu`事件被触发时把它设置成显示。当时销毁网格面板的时候，你不得不手动地把菜单配置成销毁。如果在一个需要渲染多个网格面板和一些菜单的应用程序上推断这些任务，就可以清晰地看到你需要在这一工作中引入多少重复代码。在写代码之前，让我们花一点时间分析一下问题，并找到可能的最好解决方案。

要减少重复代码的风险，你需要创建网格面板的扩展，它需要能够自动地处理菜单项的实例化和销毁。为了使这个扩展更健壮，还可以给它加哪些功能呢？

首先要考虑到事是`RowSelectionModel`和`CellSelectionModel`的获取方法和设置方法的不同选择。`RowSelectionModel`有`selectRow`和`getSelected`，而`CellSelectionModel`有`electCell`和`getSelectedCell`。如果扩展能够处理网格面板选择模型的这一变化，那会很棒。这样的功能可以减少应用层的大量代码。

考虑类的设计时必须要想想实现的多种可能方法。比如说，需要能够传入会被转变成`Menu`实例的配置对象：

```
Ext.create('MyApp.grid Panel', {
    //……（其他配置选项）
    menu : {
        items : [
            { text : 'menu item 1' },
            { text : 'menu item 2' }
        ]
    }
});
```

或者能够传入menu.Item配置对象的数组：

```
Ext.create('MyApp.grid Panel',
    //……（其他配置选项）
    menu :[
        { text : 'menu item 1' },
        { text : 'menu item 2' }
    ]
});
```

或者一个作为menu配置的Ext.menu.Menu实例：

```
var myMenu = Ext.create('Ext.menu.Menu', {
items : [
    { text : 'menu item 1' },
    { text : 'menu item 2' }
    ]
});
Ext.create('MyApp.grid Panel', {
    //……（其他配置选项）
    menu : myMenu
});
```

有了这样的灵活性之后，子类实现就符合框架的思想了，并且也使得子类可以被超过一种的方式来使用。把这个记在脑子里，对于在应用程序中设计类非常重要。当你对需要解决的问题有了清晰的理解之后，就可以构造自己的第一个Ext JS扩展了。

13.3.2　扩展GridPanel

扩展GridPanel需要用到Ext.define。下面的代码清单包含了将要创建的扩展的模板。最好把这段代码放到独立的文件中，并被HTML包含进去。因为没有使用加载器，你可以把它命名为ContextMenuGridPanel.js。

代码清单13-5　网格面板的扩展

```
Ext.define('MyApp.grid.ContextMenuGridPanel', {
    extend : 'Ext.grid.GridPanel',
    alias : 'widget.contextmenugrid',

    constructor : function() {                          ❶ 定义构造器
        this.callParent(arguments);
        if (this.menu) {                                ❷ 构建菜单
            if (! (this.menu instanceof Ext.menu.Menu)) {
                this.menu = this.buildMenu(this.menu);
            }
            this.on({                                   ❸ 钩住 itemcontextmenu 事件
                scope           : this,
                itemcontextmenu : this.onItemContextMenu
            });
        }
    },
```

```
    buildMenu : function(menuCfg) {                    ◁┐
        if (Ext.isArray(menuCfg)) {                     ❹ 添加菜单工厂方法
            menuCfg = {
                items : menuCfg
            };
        }

        return Ext.create('Ext.menu.Menu', menuCfg);
    },
    onItemContextMenu : function(grid, model, row, index, evt) {   ◁┐
        evt.stopEvent();                                             │
        this.menu.showAt(evt.getXY());                               │
    },                                      处理itemcontextmenu事件 ❺

    onDestroy : function() {

        if (this.menu && this.menu.destroy) {          ◁┐
            this.menu.destroy();                        ❻ 清除残余菜单
        }

        this.callParent(arguments);
    }
});
```

代码清单13-5包含了扩展的代码，并提供了三个会被应用到子类的原型中的方法。第一个是constructor方法❶，它是扩展网格面板的载体。在这个方法中首先调用超类的构造器，然后判断this.menu❷是否存在。如果存在并且不是Ext.menu.Menu的实例，就用buildMenu工厂方法❹并传入this.menu引用来构建一个菜单。

在buildMenu工厂方法中要判断menuCfg是否指向一个存在的Ext.menu.Menu的实例。如果不是，需要用Ext.isArray来验证一下它是不是一个普通的数组。如果是一个数组，就需要把它封装到对象里面，以便Menu的父类Container能用来做它需要做的事情。

最后还有一点关于扩展的构造器，它能够处理注册itemcontextmenu事件的监听器❸，即onItemContextMenu❺。这个监听器负责在Ext.EventObject引用（evt）上调用stopEvent()。它会屏蔽浏览器自己的上下文菜单。然后它会指示菜单在事件自己的X和Y坐标上展示出来。

最后，onDestroy方法❻扩展了GridPanel类自己的onDestroy方法。这里，你可以在菜单存在并有destroy方法的时候，为它编写自动销毁功能。

现在有了自己的定制化扩展，继续向前吧。

13.3.3 实践你的扩展

当讨论网格面板扩展的构造器时，我们提到了实现的三种模式：配置对象的menu引用可以设置成menu.Item配置对象的数组，一个Menu的实例，或者一个为Menu实例而设计的配置对象。在这个实现中，你将会选择第一种模式，使用menu.Item配置对象的数组。这么一来可以在扩展构造器里面看到Menu的自动实例化过程。

下面的代码清单包含了实现代码。因为有不少配置要放在这里以让网格面板工作，所以它显得冗长了点儿。但是现在你应该对数据存储和网格面板相当熟悉了，所以读起来应该会相对轻松很多。

代码清单13-6 为扩展实现创建一个远程JSON存储器

```
Ext.define('MyModel', {                                      ❶ 定义模型
    extend : 'Ext.data.Model',
    fields : [
        'firstname',
        'lastname'
    ]
});
                                                             ❷ 配置远程JSON存储
var remoteJsonStore = Ext.create('Ext.data.Store', {
    autoLoad : true,
    model    : 'MyModel',
    proxy    : {
        type   : 'jsonp',
        url    : 'http://extjsinaction.com/dataQuery.php',
        reader : {
            type : 'json',
            root : 'records'
        }
    }
});
                                                             ❸ 设置点击处理程序
var onMenuItemClick = function(menuItem) {
   var gridPanel    = Ext.ComponentQuery.query('contextmenugrid')[0],
       selModel     = gridPanel.getSelectionModel(),
       selectedRec  = selModel.getSelection()[0],
       msg          = Ext.String.format(
            '{0} : {1}, {2}',
            menuItem.text,
            selectedRec.get('lastname'),
            selectedRec.get('firstname')
        );

   Ext.MessageBox.alert('Feedback', msg);
};
                                                             ❹ 配置自定义网格
var grid = {
    xtype   : 'contextmenugrid',
    store   : remoteJsonStore,
    columns : [
        {
            header    : 'Last Name',
            dataIndex : 'lastname',
            flex      : 1
        },
        {
            header    : 'First Name',
            dataIndex : 'firstname',
```

```
            flex      : 1
        }
    ],
    menu      : [                                ❺ 定义菜单配置
        {
            text    : 'Add Record',
            handler : onMenuItemClick
        }
    ]
};
                                                 ❻ 实现窗口内自定义网格
new Ext.Window({
    height : 300,
    width  : 300,
    border : false,
    layout : 'fit',
    items  : grid
}).show();
```

代码清单13-6做了不少事情来配置定制的`GridPanel`类。首先，配置了模型❶和远程的JSON存储器❷。然后，设置菜单点击处理程序`onMenuItemClick`❸，它用以在某行数据被右击的时候，显示一个警告信息框。

接下来要配置一个简单的JavaScript对象❹，由Ext JS用来配置定制化的`ContextMenu-GridPanel`类的实例。通过把`xtype`属性设置为`'contextmenugrid'`来完成这一点。注意这里是如何用简单对象配置数组来作为`menu`属性的❺。这将会被`ContextMenuGridPanel`类用来创建一个`Menu`实例，并通过注册了的`itemcontextmenu`事件处理程序来展示它。

最后，在`Window`实例❻上渲染这个组件，显示效果如图13-5所示。

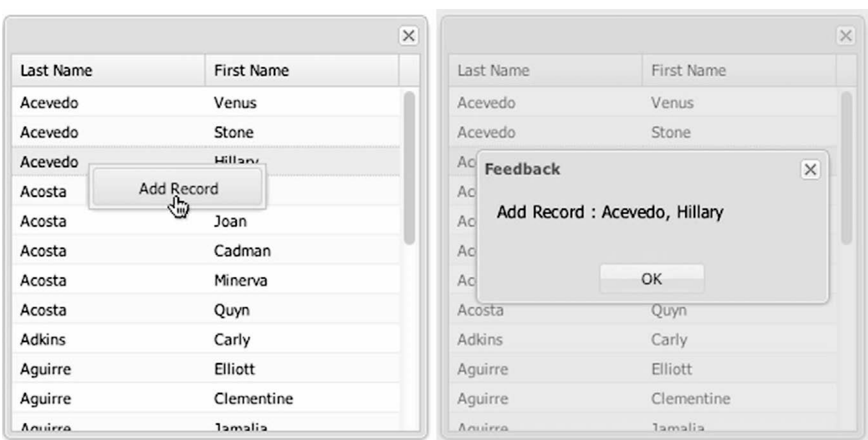


图13-5　第一次使用定制的网格面板扩展

可以看到，定制的网格面板如预期那般工作了。把这段代码导入到项目中，它会非常优雅地工作。但是在有些场景下，你可能想要选择插件而不是直接扩展网格面板来增加这些功能。比如

说，如果有一个已经在开发的或者已经在生产中的应用，怎么办呢？如何把这些扩展注入到继承链中去并且不引入风险呢？

我们再假设想要把这一功能添加到所有的数据绑定类型的视图：网格、树和通用视图。要适应这样的需求，就不得不为每一个类创建扩展，这会导致很多重复代码。

这一问题唯一可行的解决方案是插件。

13.4　用插件来救援

插件是在Ext JS 2.0版本中引入的。它解决了上述问题，支持跨部件实现功能，而不需要创建扩展。事实上，可以给组件附加任意多的插件，这使得插件更加强大。

重温组件生命周期

如果你不记得插件是什么时候创建和初始化的，现在是复习组件生命周期中的初始化阶段的绝佳时间。第3章有这一内容。

在我们深入研究怎么创建插件之前，让我们先快速聊一下插件是怎么工作的。

13.4.1　插件的剖析

插件的基本结构是简单的。它从定义类并扩展`AbstractPlugin`开始:

```
Ext.define('MyPlugin', {
    extend : 'Ext.AbstractPlugin',
    alias : 'plugins.myplugin'
    // 做些事
});
```

这一小段代码展示了使用`Ext.define`创建插件的基本步骤。创建插件，最好扩展`Ext.AbstractPlugin`类，因为它提供了必要的功能，比如在父节点自销毁之后控制插件销毁。

在这个模板中设置`alias`为`'plugins.myplugin'`。把插件的别名的前缀设置为`plugins`，这使得Ext JS类管理系统把这个类的注册交给`PluginManager`。`PgluinManager`负责通过懒对象——这很像XTypes，只是在这里它们叫作PTypes，来创建类实例。

下面是通过懒对象在一个通用的组件实例上配置插件的例子：

```
Ext.create('Ext.Component', {
    plugins : [
        {
            ptype : 'myplugin'
        }
    ]
});
```

在这段代码中创建一个Ext.Component的实例，并把它的plugins属性设置为有一个对象的数组。这个对象有一个ptype属性，被设置为'myplugin'。当一个组件快要结束初始化阶段的时候，它会通过PType捷径来创建一个定制插件的实例。

扩展AbstractPlugin被Sencha核心开发团队认为是最佳实践。定制插件的时候需要用到它。

13.4.2　开发一个插件

我们已经剖析了插件类。我们展示了基础内容，即如何注册一个插件，并通过一个简单PType配置对象来配置它。接下来，我们利用这些知识在下面的代码清单中创建定制的ViewContextMenu插件类。

代码清单13-7　定制的视图插件

```
Ext.define('MyApp.plugin.ViewContextMenu', {          ❶ 定义ViewContextMenu插件
    extend : 'Ext.AbstractPlugin',
    alias  : 'plugin.viewcontextmenu',                 ❷ 设置PType别名

    init : function() {                                ❸ 定义init方法
        if (this.menu) {
            if (! (this.menu instanceof Ext.menu.Menu)) {
                this.menu = this.buildMenu(this.menu);
            }

            this.cmp.on({                              ❹ 注册itemcontextmenu监听器
                scope           : this,
                itemcontextmenu : this.onItemContextMenu
            });
        }
    },
    buildMenu : function(menuCfg) {
        if (Ext.isArray(menuCfg)) {
            menuCfg = {
                items : menuCfg
            };
        }
        return Ext.create('Ext.menu.Menu', menuCfg);
    },
    onItemContextMenu : function(view, model, row, index, evt) {
        evt.stopEvent();
        this.menu.showAt(evt.getXY());
    },
    destroy : function() {                             ❺ 销毁实例化菜单
        if (this.menu && this.menu.destroy) {
            this.menu.destroy();
        }
    }
});
```

如代码清单13-7所示，大多数的代码和扩展中的一样，不同的是定义了一个扩展AbstractPlugin的类❶。遵循最佳实践，你设置了插件的别名❷。

接下来创建一个init方法❸，它负责侦测插件，并注册视图的itemcontextmenu事件处理程序❹。destroy方法❺负责销毁实例化了的菜单实例。

要使用这个插件，可以重用代码清单13-6中的几乎所有代码。最大的差别是配置网格面板的方式。下面的代码清单展现了实现上的不同。

代码清单13-8　配置和展现网格面板

```
// 此处为代码清单13-6中的模型、存储和菜单点击处理程序

var grid = {                                          ❶ 添加网格面板
    xtype    : 'grid',
    store    : remoteJsonStore,
    columns : [
        {
            Header    : 'Last Name',
            dataIndex : 'lastname',
            flex      : 1
        },
        {
            header    : 'First Name',
            dataIndex : 'firstname',
            flex      : 1
        }
    ],
    plugins : [
        {
            ptype : 'viewcontextmenu',                ❷ 配置插件
            menu  : [
                {
                    text : 'Add Record',
                    handler : onMenuItemClick
                }
            ]
        }
    ]
};
// 显示风格面板的窗口代码
```

代码清单13-8中配置了一个通用的网格面板xtype对象❶，它使用了前面配置的自定义的ViewContextMenu插件，并且通过Ext.Window来展示它。在这个实现中，通过plugins引用只配置了一个插件。如果有多个插件，那么就需要配置成插件数组❷，如plugins:[plugin1, plugin2, etc]。

在屏幕上渲染的话，会得到和网格面板扩展同样的功能，如图13-6所示。

如果你想要阅读其他更为复杂的插件的源代码，可以在Ext JS SDK的examples/ux文件夹下找到一些例子。我们这里将要提到的两个插件是由本书作者（Jesus）在早先3.0版本中创建的，现在由Sencha的开发团队维护。

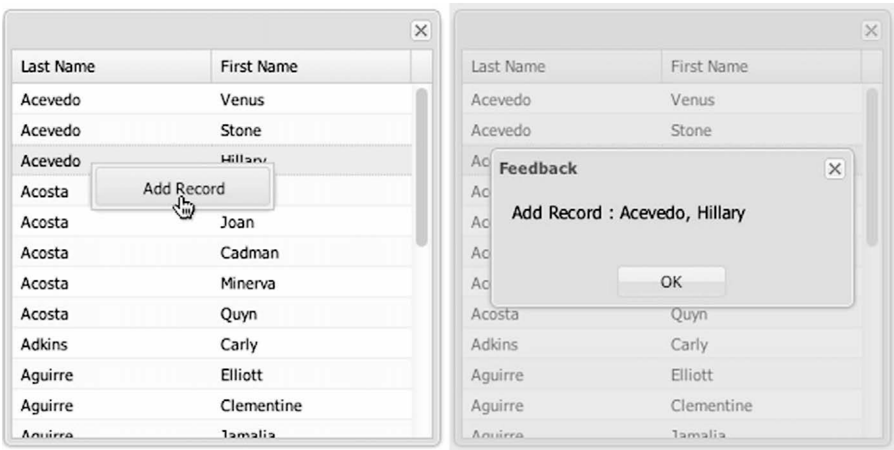


图13-6　第一次使用定制的网格面板扩展

第一个是TabScrollerMenu（TabScrollerMenu.js），如图13-7所示。这个插件给滚动选项面板增加了一个菜单，并允许用户选择并聚焦到一个选项面板上，这比滚动更容易操作。要想看这个插件是如何工作的，可以在浏览器中打开<your extjs dir>/examples/tabs/tab-scroller-menu.html。

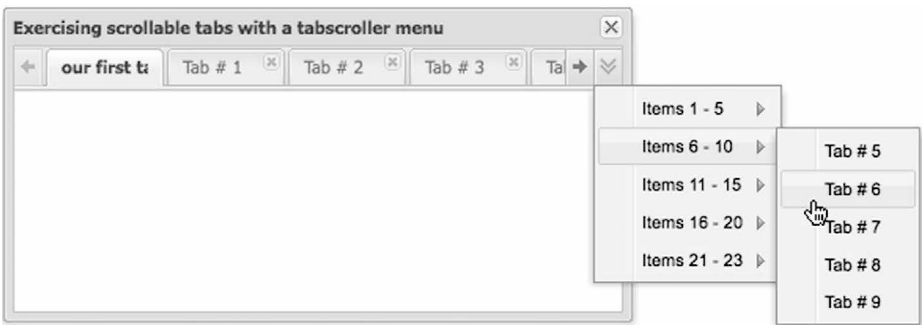


图13-7　TabScrollerMenu插件

第二个是ProgressBar分页工具条（ProgressBarPager.js），它在分页工具部件上添加了一个有动画效果的进度条，这使得分页工具条更好看了，如图13-8所示。要看这个插件如何工作，浏览器打开<your extjs dir>/examples/build/KitchenSink/ext-themeneptune/#progress-bar-pager。

我们就这样结束了插件的探索。你现在已经知道如何创建插件来给项目增强功能。如果你有使用一个插件的想法，但是不知道它是不是已经被实现过了，请访问Ext JS论坛：http://sencha.com/forum。有专门的一个板块是来讨论扩展和插件的。社区成员在那里发布了他们的作品，有一些是完全免费的。

Sencha拥有可观数量的用户自定义扩展，且可在其市场上下载到。你可以访问http://market.sencha.com。

13

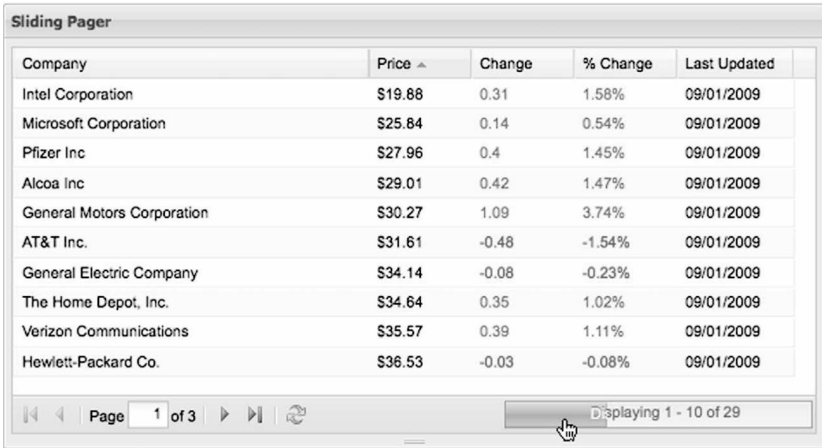


图13-8　一个添加在页面工具栏上的动画进度条插件

接下来我们要讨论Ext JS的类加载系统。开发应用之前你绝对需要这些信息，相信我们！这会为你节约大量的时间。

13.5　使用 Ext JS 加载器的动态加载类

Ext JS 4的新特性之一就是动态类加载系统，它利用了依赖模型。开发应用的时候有多种选择可以考虑，每一种选择都各有优缺点，我们将逐一探讨。

13.5.1　动态加载一切

Ext JS提供了一个选择，就是可以动态加载几乎所有的JavaScript文件。动态加载框架的文件，这样就能够快速地渲染初始页面，这对于基于因特网的应用是非常理想的。

要做到这一点，你需要引入ext-all.css和ext-debug.js文件。文件开头需要像这样写：

```
<link rel="stylesheet" type="text/css"
       href="js/ext4/resources/css/ext-all.css" />
<script type="text/javascript" src="js/ext4/ext-debug.js"></script>
```

加载ext-debug.js会加载Ext JS基础上的调试版本，这一调试版本包含了基本的类系统、工具类（如可观察类和元素类），以及加载器的框架。这意味着框架的剩余部分不会在内存中；因此，它们也需要加载。要验证这些，需要在屏幕上渲染一些东西。

下面是渲染一个`Ext.Window`的代码：

```
Ext.onReady(function() {
    Ext.create('Ext.window.Window', {
        height : 100,
        width : 100,
        html : 'I loaded dynamically.'
```

```
    }).show();
});
```

访问这个页面就会看到页面上渲染了Ext JS的窗口，但是想要理解发生了什么，你就要看底层的逻辑了。要做到这些，可以用类似Firebug这样的调试工具（参见图13-9）。



图13-9 Firebug的控制台视图，展示了Ext JS抛出的警告信息以及动态加载类的日志

当读取加载类的文件名时，你开始发现Ext JS在用有点过时的方式加载类，这和依赖系统有关。图13-10展示了Ext JS的`window`继承模型，这会帮助你更好地理解这个类的加载。

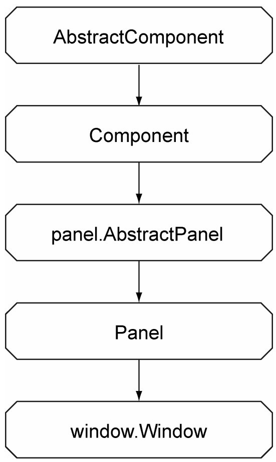


图13-10 Ext JS窗口继承模型

这段代码实例化了一个`Ext.window.Window`实例，然后看到像`Panel`这样的一些类被加载了。从图13-10中可以发现一个逆向的类加载模式，这是由于`Window`需要`Panel`，而`Panel`需要`AbstractPanel`。Ext JS加载了`Window`并读取它的依赖，然后加载`Panel`。然后，它又会读`Panel`的依赖并加载`AbstractPanel`，等等。

早些时候，Ext JS警告说它是同步地加载`Window`类。它警告你的原因是它通过同步的XHR来加载类，这意味着请求是阻塞的。Ext JS执行代码去创建一个窗口，并处理所有的需求，但是在这个时候没有其他的JavaScript代码可以执行。

在想要加载框架的一大堆代码时，这种加载JavaScript类的方式会显得有点慢。我们建议在使用框架的一个最小形式，比如用于快速渲染一些（如登录窗口等）内容时使用这种方式。如果登录成功了，你可以去触发加载需要的Ext JS类文件。

关于这一模式我们有一点需要提醒注意，这种模式下Ext JS同步加载的JavaScript是不能被调试的，因为通过XHR加载的脚本文件是在运行时执行的。这个问题是有一个解决方案，但是你需要做更多的工作。

13.5.2　应该按需加载

前面你在`Ext.onReady`的框框内创建了`Window`实例。虽说这完全可行，但却触发了框架类的同步加载。这个问题的解决办法是告诉Ext JS需要在`Ext.onReady`之外使用`Window`类。

下面是这么做需要的代码：

```
Ext.require('Ext.window.Window');
Ext.onReady(function() {
    Ext.create('Ext.window.Window', {
        height : 100,
        width  : 100,
        html   : 'I loaded dynamically.'
    }).show();
});
```

在`Ext.onReady`调用外面增加一个`require`语句，告诉Ext JS立即加载`Window`类所有的依赖，这么做在某种意义上使得调试更容易了，并能够让Ext JS不再对你“咆哮”了。这意味着需要为会用到的所有类调用`require`语句。图13-11展示了Firebug的HTML视图的一个截屏。

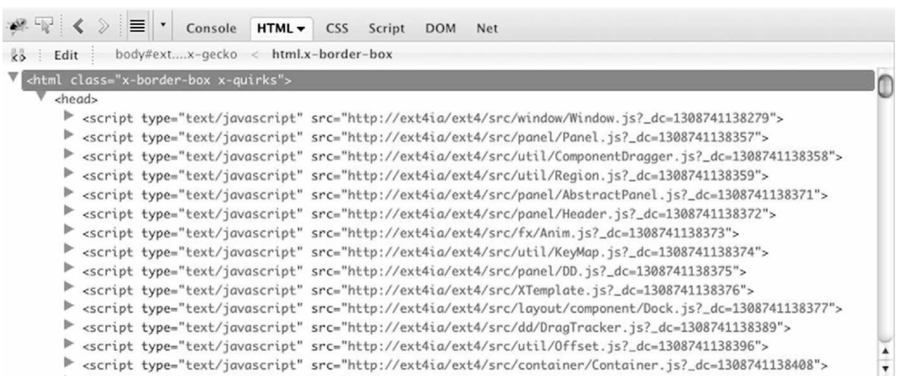


图13-11　Firebug的实时HTML视图，呈现了Ext JS类的动态加载

在图13-11中可以看到，很多的脚本标签被加到了文档开头部分。因为在`Ext.onReady`之前添加了`require`语句，它便注入了这些脚本标签，这就使得Ext JS可以通过传统的脚本标签来加载类了，而不需要执行这些脚本。这种技术最大的好处是让你能够调试Ext JS的JavaScript。

这种使用Ext JS的方式是解决调试问题的最好选择了。这是因为类是单独被加载的，所以可以把问题隔离到具体的类上。但是它很慢。在本地的开发环境上，我们看到加载一个页面的时间超过1秒。它慢的原因是一个个地加载类文件的话会有不少的请求次数。这显然不会是快速应用开发周期的最佳模式。

你需要修正一下这个方案来补救这一场景。

13.5.3　采用混合的方案

到目前为止，你已经看到了两种使用加载器来加载Ext JS类文件的方法。两种方法都有各自的优缺点。如果是在内部网络上，即网速不是问题，那么使用一个混合的方案可以达到两全其美。你可以在一次请求中加载所有的Ext JS，然后动态地加载自己的类文件。

为了研究这种混合方案，我们会提供在第13章的示例目录下的一个极度简约的应用。目录结构如图13-12所示。当观察目录结构的时候，注意文件是根据命名空间组织到硬盘上的，从MyApp目录开始。

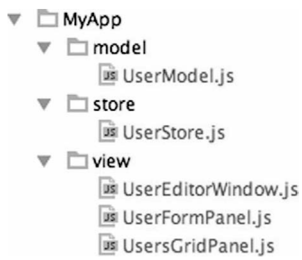


图13-12　我们示例应用程序的文件夹结构

这个应用程序包含了一个相互依赖的模型，该模型既需要Ext JS的类，还需要命名空间内的类。图13-13从高层次上审视了依赖模型。

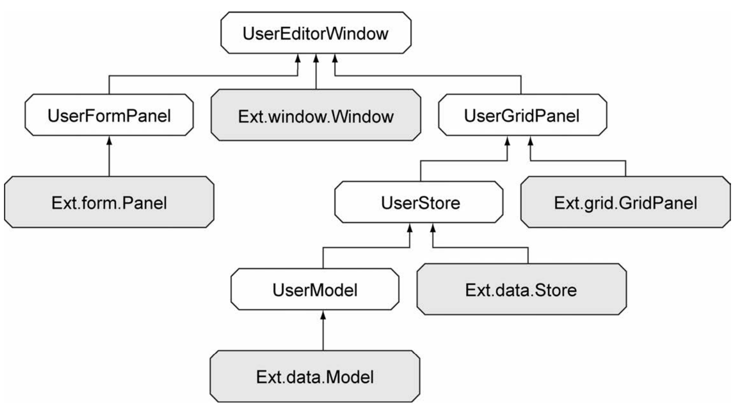


图13-13　需要开发顶层的带命名空间的类

13

这一相互依赖的模型表明了Ext JS会兑现配置的应用需求。并且，你不需要为应用代码写单个脚本标签。这是使用这种方式最大的好处。

要使这种混合模式正常发挥作用必须要有两步操作，第一步是引入ext-all-debug.js来替代ext-debug.js：

```
<script type="text/javascript" src="js/ext4/ext-all-debug.js"></script>
```

引入ext-all-debug.js可以一次加载整个Ext JS框架，这会减少超过一半的页面加载时间。这一方式的缺点是，默认情况下Ext JS的类加载系统在这个版本的框架中是关闭的。

你需要开启类加载系统，并指示它应用程序代码在何处开始。需要加入以下的代码来做到这些：

```
Ext.Loader.setConfig({
    enabled : true,
    paths : {
        MyApp : 'js/MyApp'
    }
});

Ext.require('MyApp.view.UserEditorWindow');

Ext.onReady(function() {
    Ext.create('MyApp.view.UserEditorWindow').show();
});
```

这段代码调用了`Ext.loader`单例上的`setConfig`方法，传入了一个对象来开启加载器，并设置应用程序的路径。然后，告诉Ext JS去获取`MyApp.view.UserEditorWindow`类，然后魔力发生了！

首先你会注意到这个迷你程序在屏幕上被渲染了（参见图13-14）。尽管看到程序在屏幕上出现这一点很棒，但是魔力是在底层发生的，只能通过Firebug的动态HTML标签看到。你将看到的东西非常迷人。图13-15显示所有的Ext JS代码都被加载了，但是应用程序类是动态加载的。这么做最大的好处是不需要对HTML页面上的脚本标签感到焦躁不安了。

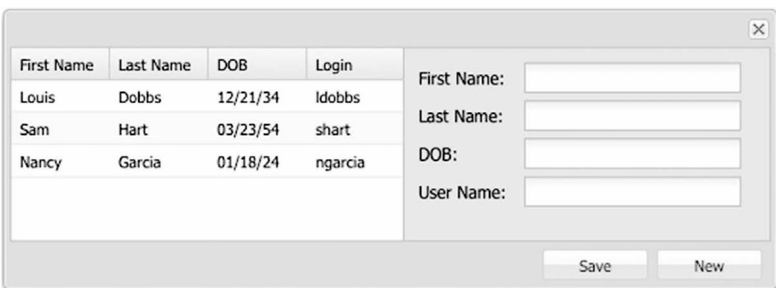


图13-14　动态加载的迷你程序

```
▶ <script type="text/javascript" src="js/MyApp/view/UserEditorWindow.js?_dc=1386253871119">
▶ <script type="text/javascript" src="js/MyApp/view/UsersGridPanel.js?_dc=1386253871179">
▶ <script type="text/javascript" src="js/MyApp/view/UserFormPanel.js?_dc=1386253871179">
▶ <script type="text/javascript" src="js/MyApp/store/UserStore.js?_dc=1386253871201">
▶ <script type="text/javascript" src="js/MyApp/model/UserModel.js?_dc=1386253871218">
```

图13-15 Firebug动态HTML标签页显示类被通过Ext JS加载，你不需要写脚本标记

到现在为止，你很可能想知道下一步怎么办。你应该选择哪一种加载模式？你确实需要思考这些问题。

事实是动态加载器不被推荐用于生产。在开发周期内，我们使用第三种模式（13.5.3节）。它通过加载所有的Ext JS到本地开发环境，并能够动态地按需加载应用代码来达到最好的性能。

对于生产，你将需要使用SDK工具在准备部署的时候来联接并缩减应用代码。我们会在第14章中探讨这些工具。

13.6 小结

本章介绍了如何用基本的JavaScript工具实现原型继承，阐述了这一继承模型是如何一步一步构建的。利用这些基础知识，你用`Ext.define`这一类定义方法重构了子类。

接下来，你使用了所有基本知识来实现Ext JS网格面板的扩展，创建了一个组合网格面板和一个菜单组件，实现了自定义的网格面板扩展，并看到了扩展组件可以多棒。

然后，你了解到扩展在需要跨部件重用的时候能力有限。缓解这一问题的方法是把网格面板扩展的代码转换成插件，该插件可以用于任意的数据视图及它的子类。

最后是实现Ext JS加载器。你亲眼目睹了如何用三种常用模式来使用加载器，并且了解了它们各自的优缺点。

在最后一章中，我们将整合本书介绍的所有知识，探索构建复杂应用的行业秘密。

第 14 章

构建一个应用

本章内容

- 像Web UI开发者一样思考
- 理解Ext JS的架构
- 编写一个基于MVC的应用
- 使用Sencha Cmd 3工具进行构建

目前为止，本书已经讲了Ext JS的组件、部件和数据的定义以及它们的使用和管理方式。我们已经了解了如何在该框架下实现各种部件，并且分别探讨了其中复杂的知识点。在上一章中，我们阐释了如何创建定制的Ext JS扩展，并深入介绍了类加载系统的工作原理。

本章，我们尝试整合在本书中学到的所有知识来创建一个实际应用。你将会在第13章的基础上进行实践，通过实现Ext JS应用包构建一个符合MVC开发模式的应用。你会渐渐了解控制器的工作原理以及它们如何响应来自应用程序类的事件，同样也将学会用Sencha Cmd（Sencha命令行工具）测试和构建产品。

在写代码前，我们先来回顾一下创建和管理Web应用的原则。无论你将来的应用是大是小，都应该时刻遵循本章描述的原则。

14.1 像 Web UI 开发者一样思考

Web应用几乎和Web本身存在了同样长的时间。在2004年谷歌的Gmail出现后，Web应用这个概念才广泛地流行起来。那之后的几年里，三个创建Web应用的主要原则被保留下来：

- Web是移动的；
- 为一个页面做设计；
- 优化服务器端服务（API）。

最快的Web应用是about:blank（空白网页），而在空白网页添加任何东西都会增加加载和运行时间。“移动优先”的思考方式，常常迫使你将信息打包成轻量级的，这是为预期的小屏幕移动设备的使用者、低配置的电脑用户、有限带宽的上网用户而考虑的。我们并不是暗示要用Ext JS开发移动应用，而是建议假设在一台旧电脑前有个用户使用你的应用的场景，那很可能就是目标

平台上的目标用户。

单页应用同样是数据驱动的。和多页的网站应用不同，服务器对单页网站仅仅提供数据，而不会准备多个HTML视图。浏览器负责根据开发者写的包含业务逻辑的代码，将接收到的数据以有意义的形式展现。整个过程不需要加载或重新加载页面。现在，各种浏览器的最新版本几乎都支持这种方式，你将会了解Ext JS如何使用这种模式。

就像这里提到的，数据是单页Web应用的关键，所以我们要更加关注它是如何提供的。最好的API是为一个目的而设计的，不要设计一个通用的接口去发送一大堆数据以备应用可能之需。API应该发送那些内容相关的、充分考虑带宽的、针对移动端优化过的有意义的、目标明确的数据。

根据这些思想，你将会看到Ext JS如何帮助创建一个优化的、数据驱动的、单页的应用。下面我们将开始讨论架构的选择。

14.2　应用的（基础）结构

让我们将Ext JS应用程序（application）看作一个应用（app）的母版。以此类推，它是所有组件被放入的地方。母版利用专有的控制器来建立视图、数据、用户交互行为之间的交流通道。

应用程序的初始建立方式会影响到它的性能、延展性，并最终决定它的可维护性。我们应该遵循规范和正确的原则，这样的话，即使是在做一个小型项目，我们也能做得很好，并且也会在项目需要做得更加严谨时让你有如神助。放心，为了节省时间而快速而随意地实现应用，绝对是通往失败的“良方”。你不可能回退并且重构应用，因为你的应用一点也不符合规则，并且是危险的产品级别软件。

开发应用的最基础原则之一是合理定义类。请确保将类合理定义。

你差不多已经可以使用Ext JS 4的所有特性来创建一个应用程序了。在这个过程中，我们将会展示很多有用的编码约定，首先就从命名空间开始讲起。还有什么步骤可以排在给伟大的产品命名更前头呢?

14.2.1　在命名空间内进行开发

正如稍后将看到的，创建一个应用程序最开始就有给应用命名这一步。给应用程序命名不一定是营销技巧，而是之后所有开发工作的基础。

你也许已经熟悉了*全局污染*（global pollution）这个词。它指的是全局的命名空间被过多的引用（变量）填满了。很常见的陷阱就是在写应用的时候使用了全局变量而导致的命名冲突。如果在不同的代码库中用同一个变量定义了两个引用该怎么办？你要因调试做噩梦了。正因此，我们总是鼓励程序员去定义一个全局变量*命名空间*，去创建他们的小生态系统。这也是面向对象编程（OOP）思想的一个基本原则。

一个浏览器环境已经被大量的全局变量污染了。表14-1分析了空白网页应用在一些现代浏览器中的被声明的全局变量。

14

表14-1　about:blank中的全局污染

浏览器	声明的变量的数量
Google Chrome 23	559
Apple Safari 6	517
Mozilla Firefox 15	184
Opera 11.64	74
Internet Explorer 7	38

在已经声明的大量变量中，你最不想做的事就是重写变量和改变整个应用的行为。此外，应用程序可能需要访问额外的API，例如地图接口、数据分析接口、其他第三方库，甚至是别的Ext JS应用程序。

当定义命名空间时，应保证它具备以下特点。

- **简洁**　开发者喜欢少打字，这样同样可以减少代码库的尺寸。
- **独一无二**　赋予一个有意义的名字，比如不要给自己的应用直接取名为“App”。

在Ext JS的世界里，就和很多其他面向结构的编程框架一样，设计类名的时候需要遵守特定的模式。在本书中，我们会提到Namespace/Package/Class（NPC）模式。

命名空间是应用程序编码开始的地方。作为根引用，应用程序的所有包和类嵌套或者子嵌套在它下面。理想情况下，整个应用程序仅在浏览器全局命名空间中加入以下两个引用。

- **Ext JS**　框架类。
- **命名空间（例如“App”）**　应用程序的类。

应用程序的MVC代码，以及单例、扩展或者其他任何代码，都应该存在于命名空间之下（图14-1）。正是由于会频繁使用命名空间，所以通常最好让名称的命名少于4个字符。还有，要确保名称有意义。例如，取名叫作App这种太一般化的名称会让人不知道你想要做什么，将来合并一些项目文件的时候也会遇到代码冲突。

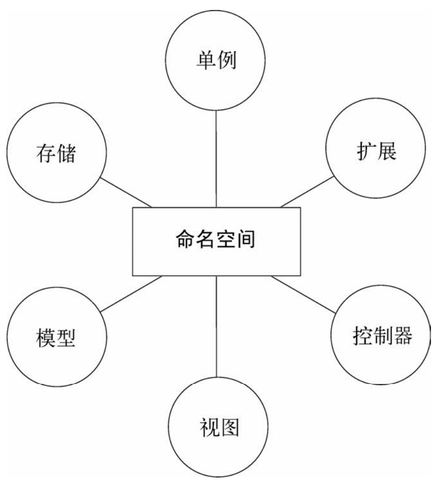


图14-1　命名空间是Ext JS应用的根节点

如图14-2所示，新定义的类会形成一个包含以下信息的名字：

(1) 应用程序的命名空间（驼峰命名法）；

(2) 包名称（小写、单数）；

(3) 类名称（驼峰命名法）。

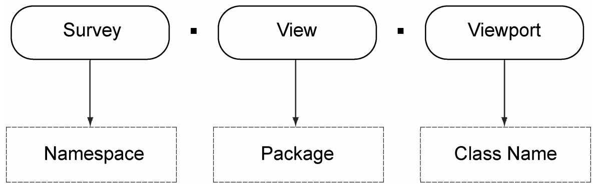


图14-2 命名空间、包、类模式

应用程序命名空间理所当然只有一个。然而，任何应用都有大量的类，并且通常有成千上百个。包正是用来有效拆分这些代码的好方法。

包最棒的地方是它们可以互相嵌套，如图14-3中展示的一样。包的第一层一般都是以下7种选择之一：

- ❑ Model
- ❑ View
- ❑ Controller
- ❑ Store
- ❑ Ux
- ❑ Util
- ❑ Component

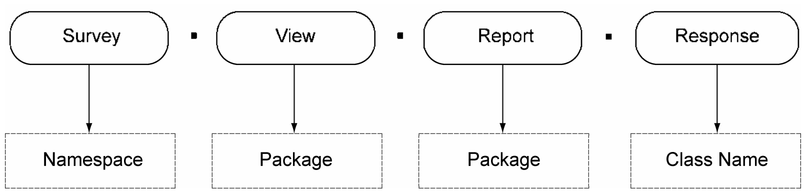


图14-3 嵌套的包

所有嵌套更深的包应该反映应用的业务逻辑。你会想要按照逻辑板块来切分应用，它们会反映应用程序模块、子模块或者抽象层次。

NPC模式命名约定不仅仅涉及命名。这个模式也定义了一个类的文件和文件夹结构。这是Ext JS 4最有价值的一个特性，即Ext.Loader起作用的地方。

14.2.2 动态依赖加载

在Ext JS 4之前，我们需要为项目运行需要的JavaScript文件硬编码脚本标记，这一直让人很

头疼。我们刻意地称为文件而不是类，因为这一特殊的情境阻碍了开发人员将每一个类分别写到各自地文件中去。在这种情况下，一个较大的应用程序会有大量含很多脚本标记的HTML文件。

Ext JS 4内置了Ext.Loader，这个有用的特性允许自动包含依赖关系。它的功能有如下两方面：

- 在需要某个类的时候进行按需加载；
- 启动运行加载明确定义的依赖关系。

根据需要加载类（动态加载）与其说是一个很酷的特性，还不如说更像一个灭火器，这是所有应用程序都很依赖的特性。它会侦查应用程序中的失败情况，使应用程序知道遗漏了哪些必要的类，并同步地加载遗漏的类文件。这一事件还会报告给浏览器的控制台，使你可以返回到代码中以合理设置依赖关系。

新的类系统提供了一些方法，可以直接或者间接定义依赖关系。一个共同点是，它们都由`Ext.define`配置。让我们来看看。

- `requires`：类定义需要的依赖（阻塞）。
- `uses`：类实例化需要的依赖（非阻塞）。
- `controllers`：应用程序使用的控制器。
- `models`：控制器使用的模型。
- `views`：控制器使用的视图。
- `stores`：控制器使用的存储。
- `extend`：在`Ext.define`中被扩展的类。
- `override`：在`Ext.define`中被覆盖的类。

为了Ext.Loader可以加载依赖关系，需要合理地存储类文件。在默认情况下，加载器会将类名扩展成完整形式，以点来分隔名称的不同部分。这就意味着每个包会成为一个子文件夹，每个类名会被看作一个以.js作为后缀的文件。但也有例外，最上面一层包，更准确地说就是你的命名空间（在图14-4例子中是`Survey`），将会被转换成预先定义的文件名：app。图14-4展示了这个过程。

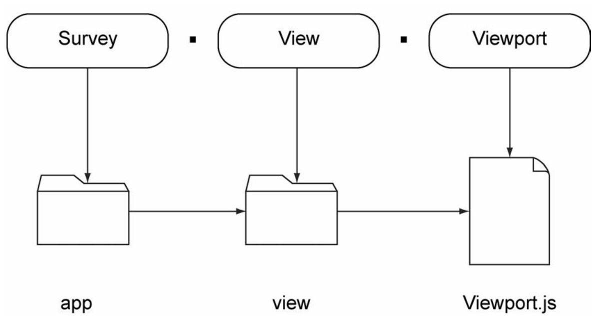


图14-4　类名转换成文件夹和文件的习惯

牢记这些原则，你会发现创建一个基本文件和文件夹结构非常简单。图14-5展示了在本章将创建的应用中使用到的文件结构。

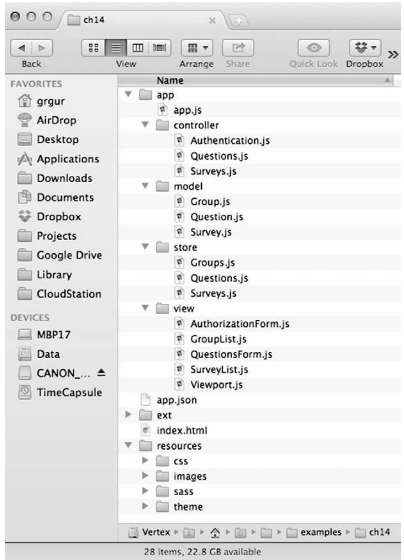


图14-5　按照命名规范定义的文件和文件夹结构

打开一个启用了Ext.Loader的应用程序需要访问Web服务器的权限。因为它利用了XHR（Ajax），而且浏览器不允许XHR对本地的文件系统的请求。但是有一个小技巧可以让浏览器即使在没有Web服务器的情况下发送Ajax请求，这就是打开Google Chrome浏览器，并将其安全防护关闭，同时打开index.html作为正常的本地文件，这样就搞定了。

关于关闭Web安全防护的警告

关闭Web安全防护，浏览器会很容易受到恶意脚本攻击。我们建议只在应用程序测试和调试时关闭Web安全防护。请警惕在浏览其他网页的时候可能涉及的风险。

以下是在不同的操作系统下关闭chrome的安全防护的方法。

Windows（命令提示符）：

```
chrome.exe -disable-web-security
```

Mac OS X（终端）：

```
open -a Google\ Chrome - - args - - disable-web-security
```

Linux（终端）：

```
chrome -disable-web-security
```

我们会在之后创建MVC应用程序的时候深入讨论依赖关系。但是在那之前，我们来看下将要做的应用程序是什么样子。

14.3　开启 Survey 应用

每一个应用都起源于一个想法，然后再设计一系列的线框图。在本章接下来的内容中，你会始终围绕一个想法，以线框图为基础构建一个应用程序，然后测试最后的结果。

我们这个Survey项目的想法是做一个调查传递平台，此处将之称为“Survey”。它唯一的目的就是为经过身份验证的的用户展示动态生成的表单、捕获输入，并同步数据源。关于这个想法，你需要的理解的概念包括：

- 模型和存储之间的CRUD操作；
- 通过关联进行数据传递；
- 以数据为基础动态生成组件；
- 用Sencha Cmd打包。

14.3.1　从想法到代码实现

Survey是一个数据驱动的MVC应用程序。我们花点儿时间来研究它的工作流：

(1) 用户需要身份验证来接受可应用的调查表和调查数据；
(2) 一个用户可以访问一条及以上的调查表；
(3) 调查问题划分成逻辑组；
(4) 调查问题一次仅在一组中出现；
(5) 用户可以根据自己的需要进入调查表或者组；
(6) 一旦有新数据填入，输入数据就立即被保存；
(7) 依据从服务器得到的数据动态生成表单。

依据这些指导方针，你会创建一个功能全面的应用程序。而且，你会有大量的领域可以通过新特性来实践并提高。图14-6展现了最终产品。

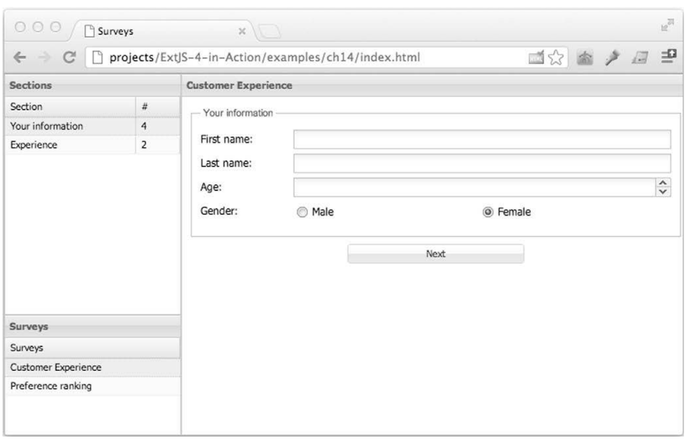


图14-6　Survey应用完成后的样子

建立用户界面就像烹饪，你花费了至少一半的时间在准备，把所有的原料和餐具集在一起，当要使用的时候随手一拿即可。我们希望你已经迫不及待要开始创建这个应用程序了。

在开始前，请按照图14-7所示的步骤——11-SAW（11-step Sencha Application Workflow）——一步一步操作。这是一个典型的开始：创建一些文件夹和文件。但是，如果我们说不需要自己手工去做开始的三个步骤，所有这些都可以用其他方式自动生成呢？你应该猜到了：Sencha Cmd可以帮你做。

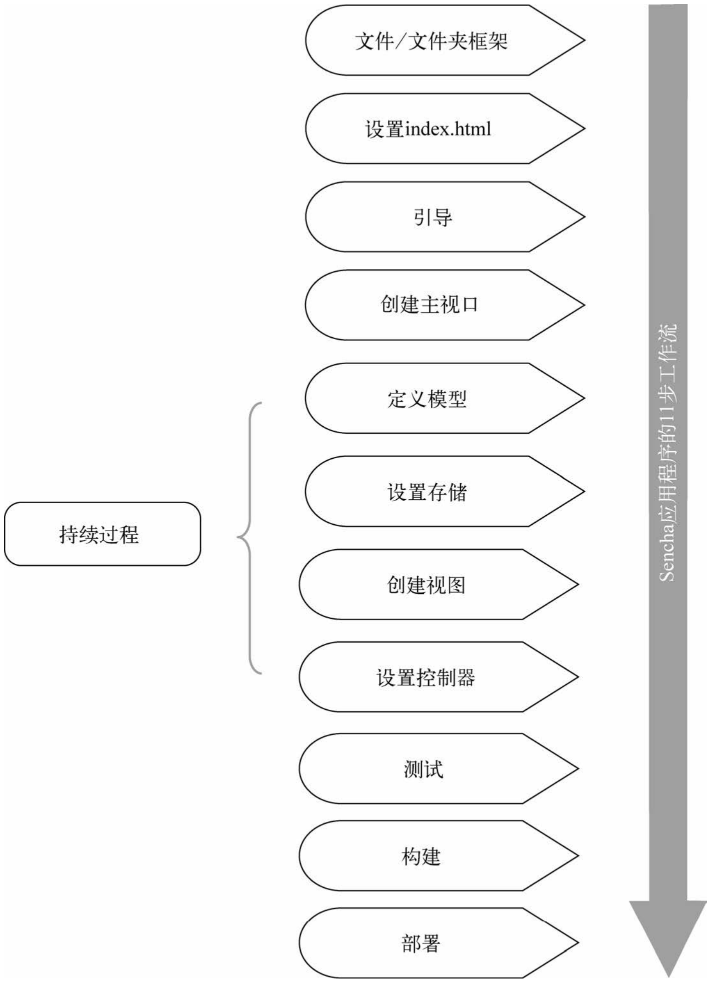


图14-7 11-SAW流程图

14.3.2 用Sencha Cmd加快开始的步伐

Sencha Cmd是一组命令行工具，目的是帮助开发者快速生成一个应用，添加模型，视图或控

制器，并且，最重要的是创建定制的应用构建。Sencha Cmd不是框架自带的，所以要从Sencha的网站上下载。

为了充分利用Cmd，你需要做以下事情：

(1) 下载和安装一个JRE（Java运行环境）6或者更高的版本；

(2) 下载和安装Compass CSS创作框架（以及根据需要下载其他的相关依赖，例如Ruby）；

(3) 下载和安装Sencha Cmd；

(4) 下载和提取最新的EXT JS SDK包（4.1.2或者更新版本）。

Sencha Cmd版本

请注意Survey应用程序是用Sencha Cmd v4.0.0.203构建的。用其他版本的Sencha Cmd来构建的话，在功能和配置上都有可能不同。

你需要做的第一件事情就是生成应用。打开命令行终端，进入包含EXT JS SDK包的目录。然后，运行以下命令：

```
sencha generate app Survey /path/to/Survey
```

这是此刻你在SDK目录下要做的所有事情。你可以导航至/path/to/Survey这个路径下（将/path/to/修改为自己电脑上实际到达Survey的路径，但是请确保没有把自己的项目文件夹建到Ext JS SDK目录中去），然后看一看（参见图14-8）。

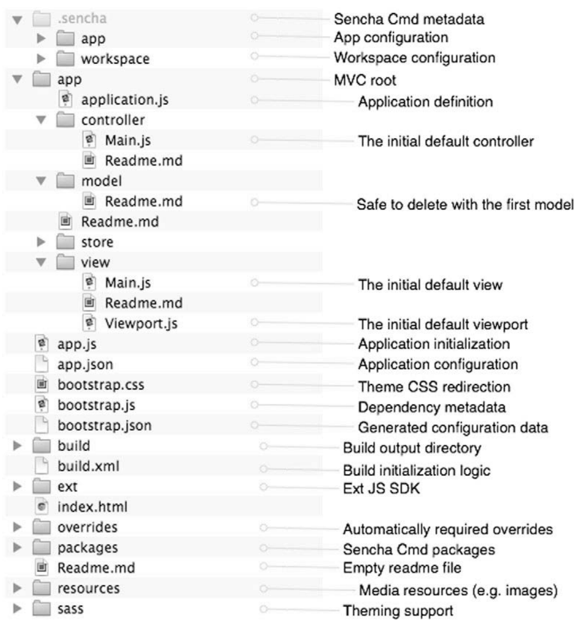


图14-8　Sencha命令行生成项目的框架

在图14-8中可以看到，Sencha Cmd确实为我们做了不少工作。

- ❑ 复制了只包含必要信息的Ext JS SDK文件（不复制文档和示例）。
- ❑ 创建一个功能齐全的index.html文件。
- ❑ 创建app.js，它是应用引导程序的基础。
- ❑ 创建一个基本的MVC框架。
- ❑ 建立SASS（语法很棒的样式表）主题项目。
- ❑ 创建一个资源文件夹，用来存储多媒体文件，例如图像。
- ❑ 创建空的readme.md 文件，来防止文件夹在某些代码控制管理系统中消失，比如GIT系统。这些readme.md可以随意删除，但是每个文件夹都必须有一个这样的文件。
- ❑ 建立私有的、隐藏的元数据文件夹，并建立一些自动生成的配置文件。这些文件和文件夹会以注释信息来说明是否可以人为修改。

生成应用代码的好处不仅仅是帮你省下这部分工作。你想要成功开发应用程序，就必须遵守纪律，并且能够严格遵照惯例。

你在开发完这个应用程序后将会具备以上两个能力，将训练有素地行事，仅在对自己长期有益的情况下走捷径。无论在哪里，你都会遵守Sencha的应用开发惯例。

现在生成了一些内容，让我们看看能用它们做什么。接下来，我们过一遍应用程序的初始化文件，并且判断想在模型中使用哪些数据格式。

14.3.3　引导Survey项目

如果你已经在电脑上完成了以上步骤，并且已经生成了自己的应用了，那么可能已经悄悄看了index.html（代码清单14-1）和自动生成的JavaScript文件。我们鼓励每个人都将整个流程走一遍。因为这样做可以让大家真正明白生成的应用到底包含哪些组成部分，就像下面的代码清单展示的那样。

代码清单14-1　生成的index.html的源代码

```
<!DOCTYPE HTML>
<html>
<head>
    <meta charset="UTF-8">
    <title>Survey</title>
    <!-- <x-compile> -->
        <!-- <x-bootstrap> -->
            <link rel="stylesheet" href="bootstrap.css">
            <script src="ext/ext-dev.js"></script>
            <script src="bootstrap.js"></script>
        <!-- </x-bootstrap> -->
        <script src="app.js"></script>
    <!-- </x-compile> -->
</head>
<body></body>
</html>
```

❶ HTML5文档类型
❷ Sencha Cmd应用引导标志
❸ 提供默认的CSS表单
❹ 指定最小SDK包
❺ Ext JS依赖映射
❻ 应用程序初始化

这里建立新Ext应用程序的最小但能够满足现今功能需要的索引页面。你将首先注意到HTML5文档类型❶。它不仅仅是帮助节省了一些字节，而且帮助Ext JS充分利用最新的浏览器。

Sencha Cmd同样也创建了一些注释语句，看起来像XML中的起始和终止标记，其实它们就和XML中的这些标记功能一样。隐藏在注释标记中的`<x-compile>`❷表示这块区域包含特殊逻辑，并且应在命令行编译阶段被修改。在`<x-bootstrap>`❸之间的语句定义引导Ext JS时需要的CSS和JavaScript文件。

bootstrap.css文件❹重定向到将要被使用的主题。这个文件不可以编辑，且应该是自动生成的并指向主题样式的CSS。Ext JS通过ext-dev.js文件❺调用，加载SDK最基本最小的部分，其他部分则动态加载。动态加载的依赖关系列表在bootstrap.js❻中，该文件也是自动生成的，因此不要手工去修改它。目前为止，浏览器知道了如何给应用程序添加样式和SDK，可以安全地调用app.js了。

在`<x-compile>`中的代码块仅仅在开发环境中才有效。本章的后面我们将会讨论根据对性能的考虑，会改变这一代码块的应用程序构建步骤。

什么是压缩？

互联网，作为一个数据分发媒介，能够以超出预期的质量提供给终端用户使用。减少传输数据量可以加快信息传输，减少使用的带宽，从而可能降低顾客的上网花费。在不影响功能的前提下，移除JavaScript上没有必要的部分的过程就叫压缩。这些没有必要的部分有空格、注释、换行符等，但压缩也可以修改引用名字，甚至更高程度的修改。尽管压缩过的代码可读性差，但是作为密码学的一种形式，压缩并不会由此变得让人迷茫与困惑。

对我们而言，我们不会依赖index.html，因为它不在Sencha Cmd app-generation过程生成的内容之内。index.html为Web页面做什么，同样app.js就为Ext JS应用做什么。它是执行所有其他代码的起始点。让我们来看看应用是如何通过以下代码被引导的。

代码清单14-2　生成的app.js源代码

```
Ext.application({                          ❶ 应用程序初始化
    name: 'Survey',                         ❷ 应用程序和命名空间名
    extend: 'Survey.Application',           ❸ 从Survey.Application中继承
    autoCreateViewport: true                ❹ 立即显示视口
});
```

`Ext.application`❶是一个特殊的方法调用，它只是加载`Survey.Application`❸类及其依赖并初始化它。它同样熟知应用程序的命名空间❷，以帮助识别哪些类可以在需要的时候动态加载。

这是复习图14-4的好时机，在那里我们讨论了如何将依赖关系转化成动态加载需要的路径。因为我们说过应用的命名空间是`Survey`，`Ext.Loader`知道`Survey.Application`位于./app/Application.js文件中。

当所有JavaScript被加载进来，并且文档已经就绪，Ext就将自动地创建视口❹。在进入视口

地讨论前，我们看看如何在Survey.Application类中配置应用程序。

代码清单14-3 应用程序定义

```
Ext.define('Survey.Application', {
    name: 'Survey',                          ❶ 应用命名（空间）

    extend: 'Ext.app.Application',
    views: [
        // 待办：在此处添加视图
    ],                                       ❷ MVC依赖

    controllers: [
        // 待办：在此处添加控制器
    ],

    stores: [
        // 待办：在此处添加存储
    ]
});
```

Survey.Application类是Ext.app.Application类的一个自然的扩展，它赋予应用程序模型、视图、控制器和存储。应用程序使用Ext.Loader来加载需要的文件，执行它们，并且分别根据它们各自的功能注册它们。

Ext.app.Application扩展了Ext.app.Controller，所以Ext.app.Application也是一个基本的控制器。这两个类的几乎所有配置选项都一样，只有name属性不同。继续看下去，本书将会告诉你更多关于控制器的事情。

你会在代码清单14-3中发现两点我们已经知道的事情。

- ❑ name属性❶在Survey.Application和Ext.application调用中被重复声明（代码清单14-2）。这更多是站在组织代码的角度考虑，而不是为了功能上的考虑。Survey.Application定义中完全可以忽略这一要求，但是Ext.application调用中名称属性声明是必要的。
- ❑ 模型在生成的类中没有被列出❷。不过，应用程序需要知道它们，你在之后过程中会把它们加进去。

你会在这个类中加入模型、控制器和存储。只要视图、模型和存储在应用中是全局性的，或者在应用程序的不同部分（模块）之间被共享，它们就要被列出来。既然这样，就把视图和存储绑定到相应的控制器配置中去，以便生成更加结构化和模块化的应用程序。

视口

有一种视图与众不同，即应用程序视口。视口是应用程序中所有视图的关键点。视口窗之于单页Web应用犹如<body>之于HTML元素。视口常常带有核心的导航元素以及通知控制台，它就像应用程序的所有主要视图的支架。一个页面只可以有一个视口。

如代码清单14-2所示，当文档准备就绪可以进行DOM操作时，视口就会被自动创建。Ext JS会自动搜索view.Viewport类。在这个例子中是Survey.view.Viewport类。所以，让我们一

起写这个类。

在写代码之前，请先花一点儿时间考虑一下你想获得什么。回头看看应用程序需求，你会想起只有注册用户可以访问Survey，这意味着需要提供注册表单。由于登录了的用户在某些情况下会想要退出，因此请为视口设计卡片布局，以加快注册表单页和回退的页面之间的切换。下面是创建视口的代码（这仅仅是一个存根）。

代码清单14-4　视口存根

```
Ext.define('Survey.view.Viewport', {
    extend : 'Ext.container.Viewport',
    alias  : 'widget.vp',

    layout : {
        type : 'card'
    }
});
```

通常情况下`Viewport`是对`Ext.container.Viewport`类的扩展。这是一个特殊的类，它自动地占用了文档体全部可用的长度和宽度。这样的视口不允许滚动，组件如果想要滚动效果的话需要自行开启。最后，为了将来方便访问你需要设定`vp`作为XType，并且将布局设置为`'card'`。

注意　Sencha Cmd已经创建了这个文件的一个示例，你可以在代码清单14-3的基础上修改，而不必自己重头创建。另一个自动生成的视图在Main.js里，你可以安全地删除它，因为不会用到它。

在完成了11-SAW的4步之后，让我们检查下目前的成果，并为下一步骤（即数据建模）做准备。

14.3.4　数据驱动的应用程序模型

如果在浏览器中运行目前的应用程序，你只会看到一个空白的页面。这是一个好的信号，因为它意味着已经可在建立数据模型后设计视图了。毕竟，你将设计一个数据驱动的应用程序。

在这里，11-SAW过程指示你来定义模型。考虑下这个问题，之前预期想要的数据模型是怎样的？很明显，你会需要一个Survey列表，因此还要定义Survey的外观。此外，Survey是由一个一个的问题组成，这意味着还需要一个模型用来定义问题。并且问题是分组的，你还要定义不同的组。以上这个简短的头脑风暴揭示了模型会有：

- Survey
- Group（组）
- Question（问题）

在卷起袖子干活前，再花点儿时间思考一下。你只有触及一个想法的本质时，才可能以最简洁的方式去创造。你将要建立的模型都会和另外一个连接起来。一个Survey中包含各种组，每个组有

一个或者更多的问题。因为它们被很好地连接起来，为什么不使用个模型关联呢？当你想要一次性下载所有数据，并且由最顶层的模型将数据项分别推送到相应的模型时，关联会很有帮助。

乍一听起来觉得有点含糊。在下面的代码清单中，会创建一个数据对象的样本，帮助你获得清晰的认识。

代码清单14-5 期望的数据对象

```
[
    {
        id     : 1,
        name   : 'Sample Survey',
        groups : [                                    ❶ 添加关联键
            {
                id        : 11,
                name      : 'Sample Group',
                survey_id : 1,                                    ❷ 设置外键
                questions : [
                    {
                        id        : 111,
                        survey_id : 1,
                        group_id  : 11,
                        question  : 'Sample Question',
                        config    : {                                    ❸ 配置问题字段
                            xtype : 'textarea'
                        }
                    }
                ]
            }
        ]
    }
]
```

在只有20行的代码中，你可以知道关于数据的很多东西。Surveys会直接展示它所得到的数据，你所需要的仅仅是一个名称和识别码。Survey带有特殊的属性，即`groups`❶，该属性代表了问题组的一个数组。这也是我们知道的has many关联关系。

组是简单的。每个组都有另一个特殊的属性，即`survey_id`❷。这个属性用来将组和它对应的Survey关联起来，相当于一个数据库外键的概念。和Survey一样，group也和question有has many关联关系。

问题则有点复杂了，它具有配置一块区域所需要的所有属性。注意，`config`属性❸在`Ext.field.Field`类和其子类中几乎可以接受任何配置属性。

下一步，我们来看如何设置每个模型和它们各自的存储。

14.3.5 给应用程序增加模型

Survey模型仅由两个字段组成：

- ❑ `id`
- ❑ `name`

除了建立字段，还需要定义关联关系和数据代理。最终代码如代码清单14-6所示。

代码清单14-6　Survey模型

```
Ext.define('Survey.model.Survey', {
    extend : 'Ext.data.Model',

    requires : [
        'Ext.data.association.HasMany'
    ],
                                                ❶ 添加相关模型
    uses : [                                   <─┘
        'Survey.model.Group'
    ],
                                                    ❷ 填充组
    associations : [                               <─┘
        {
            type           : 'hasMany',
            model          : 'Survey.model.Group',
            primaryKey     : 'id',
            foreignKey     : 'survey_id',               ❸ 列出组数据
            autoLoad       : true,
            associationKey : 'groups',                 <─┘
            name           : 'groups'          <─┐
        }                                          ❹ 配置组方法
    ],

    proxy : {                                  <─┐
        type   : 'ajax',                          ❺ 添加代理配置
        url    : 'data.json',
        reader : {
            type : 'json'
        }
    },

    fields : [                                 <─┐
        {                                         ❻ 设置字段列表
            name : 'id'
        },
        {
            name : 'name'
        }
    ]
});
```

Survey和Group是有关联的，因此必须让类知道它依赖于`model.Group`类❶。一个Survey可以拥有很多个组，这意味着它有`hasMany`关联关系❷。在关联配置里，使用`associationKey`属性❸来帮助自动分组。组会存在于属性组中。最终，可以通过`groups`方法❹调用来访问组，并且用`name`配置来监听。当将这个模型类插入视图，就可以返回到这个模型类了。

建立关联可能是定义Survey模型时工作强度最大的部分。下面建立一个代理❺，这样一个`Model`实例知道存储应该从哪里获取数据以及如何完成剩下的CRUD操作。你不用在配置存储的

时候重复建立代理，因为它会自动被共享。最后，用需要的字段名简单地配置一下fields❻即可。

Survey.model.Survey现在能够知道Survey从哪里来及如何辨识其中的字段了。并且，它也知道了分组需要通过另一个模型来处理，即Survey.model.Group。

Group模型（代码清单14-7）非常像Survey，但有两个主要的不同之处：

- 不和服务器交互；
- 它从Survey模型接受数据，并推送问题到Question模型，因此它具有belongsTo和hasMany关联关系。

代码清单14-7　Group模型

```
Ext.define('Survey.model.Group', {
    extend : 'Ext.data.Model',

    requires : [
        'Ext.data.association.HasMany',
        'Ext.data.association.BelongsTo'
    ],

    uses : [
        'Survey.model.Survey',
        'Survey.model.Question'
    ],

    fields : [
        {
            name : 'id'
        },
        {
            name : 'survey_id'
        },
        {
            name : 'name'
        },
        {
            name : 'index'
        }
    ],

    associations : [
        {
            type       : 'belongsTo',           ❶ 确认与Survey模型的关联
            model      : 'Survey.model.Survey',
            primaryKey : 'id',
            foreignKey : 'survey_id'
        },
        {
            type           : 'hasMany',         ❷ 添加hasMany关联
            model          : 'Survey.model.Question',
            primaryKey     : 'id',
```

```
                foreignKey     : 'group_id',
                autoLoad       : true,
                associationKey : 'questions',
                name           : 'questions'
            }
        ],

        proxy : {                                   ⬅
            type   : 'memory'                      ❸ 配置存储代理
        }
    });
```

由于Survey和Group关联，后者需要设置belongsTo到Survey上的关联❶。Group表现得像一个中间媒介一样，它从Survey接受数据，然后转发一些给Question。因此，Question和Group有了hasMany关联关系❷。

Group都是只读的。它们也从其他模型中接受数据，正因此我们要设置proxy为内存类型❸。我们必须设置一个代理。

最终的模型已经一目了然，如下面的代码清单所示。你需要建立字段和一个belongsTo关联，将问题和组连接起来。

代码清单14-8　Question 模型

```
Ext.define('Survey.model.Question', {
    extend : 'Ext.data.Model',

    requires : [
        'Ext.data.association.BelongsTo'
    ],

    uses : [
        'Survey.model.Group'
    ],

    fields : [
        {
            name : 'id'
        },
        {
            name : 'group_id'
        },
        {
            name : 'question'
        },
        {
            name : 'answer'
        },
        {
            name : 'config'
        }
    ],
    belongsTo : [
```

```
            {
                model      : 'Survey.model.Group',
                foreignKey : 'group_id'
            }
        ],

        proxy : {
            type   : 'memory'
        }
    });
```

Question模型负责动态表单的生成。因此，它本质上由决定生成字段的配置信息属性组成。question属性包含问题标签，要把它作为Ext.field.Field子类的fieldLabel属性来使用。与此类似，answer属性包含一个新记录的值，甚至可以从服务器上抓取一个以前保存的记录。其他配置参数包括xtype，都可以在config字段中来指定。

模型在数据处理上做了很多苦力活。它们表示了数据及其字段的集合，并能够标准化和验证绑定到它们身上的数据，还知道如何通过关联关系连接其他的模型。但是模型仅仅表示单一的数据定义，还需要为所有的Model实例建立一个仓库：存储。

14.3.6　添加数据存储

所有这三个数据存储都映射到同样的定义，这归功于各自模型的详尽设置。让我们在下面的代码清单中看看存储。

代码清单14-9　Surveys存储

```
    Ext.define('Survey.store.Surveys', {
        extend : 'Ext.data.Store',

        requires : [
            'Survey.model.Survey'
        ],

        storeId  : 'Surveys',
        autoLoad : true,
        model    : 'Survey.model.Survey'
    });
```

现在你应该熟悉了数据存储的工作原理。这个例子引用了模型名称和使得从视图中引用存储变得更容易的storeId，你只需要指定一个视图到storeId，然后视图组件就能在必要的情况下实例化存储了，否则它就会重用已经存在的实例。

命名约定注意事项

模型定义了单条记录的配置，而存储定义了一系列Model实例的配置。因此，模型通常用单数命名而存储用复数，比如Survey.model.Survey和Survey.store.Surveys。

定义数据结构是很困难的事情。这需要在前期的架构设计中计划好，并在开发之前就早早地做一些重大的决定。选择正确的数据模型会影响客户端的性能、带宽消耗，并最终关系到用户体验。

一般来说，服务器端的限制会影响到客户端的数据模型。比如，服务器管理员可能仅有有限的资源来格式化现存的API，以发送Ext JS应用程序所需要的数据来工作。在这种情形下，确保折中的方案对客户端的负面影响最小。在我们的例子中，服务器管理员应该努力扔掉数据对象不必要的部分，并格式化为JSON而不是XML，使用压缩，减少递归，并使用任何其他有益于客户端的技巧。客户端计算机可能比服务端的弱，但是你不愿意让用户感觉到任何延迟。

我们对11-SAW的学习已经过半，你已经取得了很大进步。你会继续构建视图和控制器，然后就可以在浏览器上跟踪进度了。

14.3.7　创建验证表单

Survey应用的登录页面是验证表单。它的目标是让用户输入凭据，并允许他们去调查选择页面。这个视口是一个卡片布局容器，所以可以将验证表单（见图14-9）作为视图的第一个子节点。一旦用户成功登录，视口就会切换到第二个卡片，即“调查名单”上。



图14-9　验证表单设计

验证表单将被置于页面的中间。让我们看一下视图的详细信息：

- 它在屏幕上居中位置；
- 字段被封装在字段集组件内；
- 字段锚定在100%；
- 登录按钮横跨了整个可用宽度。

下面的代码清单可以让你好好复习下表单和布局的工作原理。

代码清单14-10　验证视图

```
Ext.define('Survey.view.AuthorizationForm', {
    extend : 'Ext.form.Panel',
    alias : 'widget.authform',

    requires: [
        'Ext.form.field.Text',
        'Ext.form.FieldSet',
        'Ext.Button'
    ],

    layout : {
```

```
            align : 'center',        ❶ 居中字段集
            pack  : 'center',
            type  : 'hbox'
        },

        items : [
            {                             ❷ 添加字段集
                xtype : 'fieldset',
                width : 300,
                title : 'Log in',
                items : [
                    {
                        xtype      : 'textfield',
                        anchor     : '100%',
                        fieldLabel : 'Email'
                    },
                    {
                        xtype      : 'textfield',
                        anchor     : '100%',
                        inputType  : 'password',
                        fieldLabel : 'Password'
                    },
                    {
                        xtype      : 'button',
                        anchor     : '100%',
                        itemId     : 'loginBtn',
                        text       : 'Log in'
                    }
                ]
            }
        ]
    });
```

VBox布局❶保证了项目既是垂直居中的，又是水平居中的。`fieldset`❷包括了用来输入验证用的电子邮件和密码的字段。和确认按钮一样，它们被锚定至100%的可用宽度。

> **友情提示**
>
> 如果把布局从VBox改成HBox，会有什么影响呢？

下面的代码清单给主视口添加了一个代表保护区域的空组件。

代码清单14-11 添加表单到视口

```
    items : [
        {
            xtype : 'authform'
        },
        {
            xtype : 'component',
            html : 'Protected area'
        }
    ]
```

添加代码清单14-11的代码到Survey.view.Viewport。第一张卡片是验证表单，将在视口可见之后默认显示。第二张卡片是受保护区域组件，它要在验证成功之后才出现。

注意，受保护视图是另一个不需要进一步应用验证机制的卡片。恶意用户很容易更换活跃的卡片。安全处理过程因应用程序而异，所以我们不在这里讨论它，但是对于这一点要做到心中有数。

你将很快能在浏览器上看到变化。下一步是创建第一个控制器，它会监听提交按钮的点击事件，并处理验证逻辑，最终展示受保护的数据。

14.3.8　插入第一个控制器

控制器的主要目的是建立视图、数据和用户交互之间的通信通道。控制器是基于事件开发流程的一个重要元素。它们捕获组件触发的事件，并与它们合作来确保流畅的用户体验。控制器把视图、模型、存储和应用作为4个主要的交互源和目标。此外，Application是Controller的子类。两者之间最明显的区别在于，控制器不能实例化其他控制器。这受到Application的规则影响。下面的代码清单很好地展示了控制器的通常使用方法。

代码清单14-12　Authentication控制器

```
Ext.define('Survey.controller.Authentication', {
    extend : 'Ext.app.Controller',

    views : [                                          ❶ 配置视图
        'AuthorizationForm'
    ],

    init : function (application) {
        this.control({                                   ❷ 创建点击处理程序
            "button#loginBtn" : {
                click : this.onLoginClick
            }
        });
    },

    onLoginClick : function (button) {
        // 验证处理……
        button.up('vp').getLayout().setActiveItem(1);
    }
});
```

虽然控制器能够指定视图、模型和存储作为它的依赖项，但是在这个例子中只需要指定一个视图❶，即验证表单。控制器应该只指定它们真正控制的依赖，因此不需要进一步指定任何其他的视图。

开发人员常常不知道一个控制器包含了哪些模型、视图和存储。经验法则表明，你应该把它们看作有意义的、自给自足的包。你只需要指定那些控制器离开了就没法工作的类。

正如Ext.Component有initComponent方法，控制器也有对应的init方法。它最常被用来执行控制器的control方法。与组件上的addListener方法类似，control方法负责找到和查

询的组件匹配的所有组件，并给它们分配一个监听器❷。因为控制器在视图之前被实例化，所以控制器可以很智能地和将来要实例化的组件一起工作。

事件处理程序经常在控制器内定义。按照惯例，所有由组件触发的事件都应该把触发了的实例作为第一个参数传递给监听回调方法。因此，位于控制器内的回调方法会有一个简单的方式来访问该组件的实例引用。

简单起见，这一特定的组件剥离了现实生活中的验证流程。使用目前的功能，用户能够输入他们的电子邮件和密码并点击提交按钮，控制器会把它们发送到主视口中的下一张卡片上去。你现在应该去创建这个应用程序中的其他视图和控制器了。

14.3.9 Survey视图

你现在已经熟悉了Ext JS中的MVC流程了，接下来要为剩余的类创建视图和控制器定义。视图将会相当简单，然而控制器要用到一些额外的业务逻辑来使之工作。

为了展示Survey应用的剩余部分，你需要三个视图类型。

- 调查列表：可用调查的列表。
- 组列表：选中的调查的可用组的列表。
- 问题表单：装载动态创建的调查表单字段。

下面的代码清单包含了Survey视图的代码。

代码清单14-13 Survey视图

```
Ext.define('Survey.view.SurveyList', {          ❶ 添加调查列表
    extend : 'Ext.grid.Panel',
    alias  : 'widget.surveylist',
    title       : 'Surveys',
    columnLines : false,
    store       : 'Surveys',
    cls         : 'surveylist',            ❷ 隐藏网格面板标题行

    columns : [
        {
            xtype     : 'gridcolumn',
            flex      : 1,
            dataIndex : 'name',
            text      : 'Surveys'
        }
    ]
});

Ext.define('Survey.view.GroupList', {           ❸ 定义GroupList 视图
    extend : 'Ext.grid.Panel',
    alias  : 'widget.grouplist',

    title       : 'Sections',
    columnLines : true,
    store       : 'Groups',
    cls         : 'groupList',
```

```
        columns : [
            {
                flex      : 1,
                dataIndex : 'name',
                text      : 'Section'
            },
            {
                Xtype    : 'numbercolumn',
                Width    : 50,
                text     : '#',
                renderer : function (value, meta, record) {      ◁ ❹ 计算问题数
                    return record.questions().getCount();
                }
            }
        ]
    });

    Ext.define('Survey.view.QuestionsForm', {         ◁ ❺ 定义QuestionForm类
        extend : 'Ext.form.Panel',
        alias  : 'widget.questions',

        requires : [                                ◁ ❻ 规定所有字段类型
            'Ext.form.field.Checkbox',
            'Ext.form.field.ComboBox',
            'Ext.form.field.Date',
            'Ext.form.field.Display',
            'Ext.form.field.Hidden',
            'Ext.form.field.HtmlEditor',
            'Ext.form.field.Number',
            'Ext.form.field.Picker',
            'Ext.form.field.Radio',
            'Ext.form.field.Spinner',
            'Ext.form.field.Text',
            'Ext.form.field.TextArea',
            'Ext.form.field.Time',
            'Ext.form.RadioGroup',
            'Ext.form.CheckboxGroup'
        ],

        layout : {
            type  : 'vbox',                   ❼ 将所有子项居中
            align : 'center'                ◁
        }
    });
```

为了显示可用的调查列表，这里使用了带有单列的网格面板❶。因为只需要显示名字，不需要标题行，所以把它禁用了❷。坦率地说，通过CSS隐藏标题行来移除它的。它们已然是存在的，并且标题相关的处理不会消失不在。

接下来是Groups列表组件❸，它也是个简单的网格面板，和Survey列表类似。Groups列表有一个额外的列，它填充了一个组中的问题数量的数据。获取数字的方式很有趣：通过关联关系来访问一个组的问题。调用关联关系会产生一个加载了相应记录的`Store`实例。可以使用

Ext.data.Store类上的一个成员方法getCount()❹来计算。

> **hasMany关联关系存储何时被创建？**
>
> 当访问一个Model实例的（也叫作记录的）hasMany关联的数据，正如代码清单14-13所示，Ext JS会创建一个Ext.data.AbstractStore实例，该实例是Ext.data.Store的简化版本。这个实例会和记录一起保留在缓存中供将来使用，但是也会和它一起被销毁。

更为简单的是，QuestionForm❺用作最终会被渲染进来的所有问题❻的父容器。我们计划在问题组下面添加一个按钮，以便更容易导航到下一个组，或者最终结束调查。为了使它更美观，该按钮会被水平居中显示到屏幕上。设置布局类型为vbox，并把它对齐到中心❼，是可以达到所需效果的一种简单方式。问题会伸展以占据整个可用宽度。这一部分会由控制器动态完成，我们将会在下面讲到。

更新视口

现在需要为所有新定义的视图创造一个“家”了。在Survey.view.Viewport类中，它的布局类型设置为card。第一张卡片带来了验证屏幕，它同时也初始化了Survey应用程序的屏幕，如代码清单14-11所示。在代码清单14-12中，Authentication控制器会切换到第二个嵌入在视口中的卡片，这正是放置调查相关视图的地方。

图14-6显示组列表堆叠在调查列表之上。两个列表都占据了屏幕的左侧（西侧）区域。较大的区域是为问题预留的。把它们都插入到Viewport类中是合理的，如代码清单14-14所示。

代码清单14-14　更新Viewport类

```
Ext.define('Survey.view.Viewport', {
    extend : 'Ext.container.Viewport',
    alias  : 'widget.vp',

    requires : [
        'Survey.view.AuthorizationForm',
        'Survey.view.SurveyList'
    ],

    layout : {
        type : 'card'
    },

    items : [
        {
            xtype : 'authform'
        },
        {
            xtype  : 'container',          ❶ 添加调查视图
            itemId : 'mainContainer',
            layout : {
                align : 'stretch',
                type  : 'hbox'
            },
```

```
                items   : [
                    {
                        xtype     : 'container',          ❷ 设置左栏容器
                        minWidth : 200,
                        flex      : 1,
                        layout    : {
                            align : 'stretch',
                            type  : 'vbox'
                        },
                        items     : [
                            {
                                xtype  : 'grouplist',      ❸ 添加GroupList 视图
                                flex   : 2,
                                hidden : true
                            },
                            {
                                xtype : 'surveylist',      ❹ 添加Survey 列表
                                flex  : 1
                            }
                        ]
                    },
                    {
                        xtype       : 'questions',         ❺ 添加Question视图
                        bodyPadding : 10,
                        flex        : 3
                    }
                ]
            }
        ]
    });
```

所有的视图都嵌套在一个以HBox布局的容器之下❶。容器把屏幕切分成了4个均等的垂直板块。其中3个板块用来给问题使用❺，1个版块给左边的导航条使用❷。后者是另外一种盒子类型布局的容器，但是这一次它是垂直导向的。有两个部分被组列表占据❸，另一个则被调查列表使用了❹。

如你所见，grouplist被初识化为隐藏的，以让surveylist成为焦点。这是有道理的，因为用户在继续操作之前要先选择一项调查。选择一个调查会自动地处理组列表的visible属性，重新计算高度以匹配需要的布局模式。这一自动化过程是由控制器控制的，它会把我们引向它们的设置。

14.3.10　Survey控制器

正如刚刚所看到的，视图是基本的、简单的。这是它们应该存在的方式：没有数据，并且几乎没有交互。控制器则操控视图上所有的动态内容。

你在定义Authentication控制器类的时候使用了控制器（代码清单14-4）。那让我们定义了最后两项：Surveys和Questions控制器，它们各自管理应用程序相应的领域。让我们来看看这两个控制器类。

1. Surveys控制器

调查需要的交互量很少，但是它集拢了所有经常会被使用的控制器特性。正如本章先前描述过的，控制器在很大程度上管理着依赖。它尤其需要引用Surveys和Groups相关的所有视图、模型和存储。Surveys控制器还有一个额外的任务：等待用户从Survey列表视图中选择一个调查，并在进一步的选择中展示恰当的组。让我们在下面的代码清单中实现它。

代码清单14-15　Surveys控制器

```
Ext.define('Survey.controller.Surveys', {
    extend : 'Ext.app.Controller',

    models : [                                    ◁─┐
        'Survey',                                   │
        'Group'                                     │
    ],                                              │
    stores : [                                    ◁─┤ ❶ 设置依赖
        'Surveys',                                  │
        'Groups'                                    │
    ],                                              │
    views : [                                     ◁─┘
        'GroupList',
        'SurveyList',
        'QuestionsForm'
    ],

    refs : [
        {
            ref      : 'groupList',
            selector : 'grouplist'
        }
    ],

    init : function () {
        this.control({
            surveylist : {
                select : this.loadGroups
            }
        });
    },
                                                       ❷ 监听以选择项目
    loadGroups : function (grid, record) {
        var groups = this.getGroupList(),             ◁─┘
            groupRec,
            questions;

        groups.show();
                                                        ❸ 访问GroupList引用
        groups.reconfigure(record.groups());

        groupRec = groups.getStore().getAt(0);         ◁─┘
        if (groupRec) {                                     ❹ 选择第一个组
            groups.getSelectionModel().select([groupRec]);  ◁─┘
```

```
        }

        questions = groups.up('#mainContainer').down('questions');
        questions.setTitle(record.get('name'));
    }
});
```

第一个步骤是定义依赖❶，所有模型、存储和视图都被指定了。这很合理，因为这个控制器要与调查和组交互。另外，尽管我们谈论过QuestionsForm视图，你仍计划要建立一个专用的Questions控制器，这是因为QuestionsForm会在被控制器继续访问的过程中动态地设置标题。

在14.2.2节中，我们讨论了集中设置依赖的方式，提到了控制器、模型、视图和存储。这四种类型是不同的，因为不需要指定完全的类名称。打个比方，你很容易断定Group模型的类名是Survey.model.Group。因此，只需要指定名称的最后一部分（比如，如果类名是Survey.model.package.Group，那么就是Group或者package.Group）。

在进一步学习之前，让我们先了解一下引用。引用用定义好的组件查询选择器快速地访问第一个被实例化的组件。创建一个引用以在需要的时候快速访问GroupList。引用会自动创建一个获取方法，它是一个便利方法，用来调用引用的实例。这个引用会创建一个getGroupList()方法❷，它是Surveys Controller实例上的成员。

网格选择监听器高效地调用loadGroups回调方法，后者会对用户界面做一些更改。首先，它会使用配置的引用获取方法❸使GroupList可见。把列表设置为可见，这会导致左边的列重新计算布局，GroupList和SurveyList之间最终被强制为2：1的高度比例。

接下来要重新配置GroupList，也就是说用新定义的存储来代替旧的。新的存储是在record.groups()方法被调用的时候自动生成的，并且还创建了hasMany关联关系。新的存储（或者说关联关系）就要被访问，以决定第一条记录，该记录也会被自动选择❹。这一部分很有趣。通过选择列表项，你还会触发select事件，后面会用该事件来加载问题。但是，你预留了那一块给另一个的控制器了，它是专门给问题用的。注意到分离两个控制器的细线了吗？

让我们来看Questions控制器吧！

2. Questions控制器

最全面的控制器负责渲染问题，并在用户输入或者选择的时候保存它们的值。说起来容易做起来难，但是框架在这里其实做了很多工作。看看下面的代码清单。

代码清单14-16 Questions控制器

```
Ext.define('Survey.controller.Questions', {
    extend : 'Ext.app.Controller',

    views : [
        'QuestionsForm'                          ◁─┐
    ],                                             ❶ 添加控制器依赖

    refs : [
        {
            ref       : 'form',
```

```
        selector : 'questions'
    },
    {
        ref      : 'groups',
        selector : 'grouplist'
    }
],

init : function (application) {
    this.control({
        grouplist : {
            select : this.showGroupQuestions
        },
        '#groupNext' : {
            click : this.showNextGroup
        },
        '#surveyFinish' : {
            click : this.finishSurvey
        },

        'questions field' : {
            change : this.saveItem
        }
    });
},

showGroupQuestions : function (grid, record, index) {
    var questions = record.questions(),
        form = this.getForm(),
        store = grid.store,
        isLastGroup = (store.getCount() - index) === 1,
        fields = [];

    questions.each(function (question) {
        var field = Ext.apply({
            fieldLabel : question.get('question'),
            value      : question.get('answer'),
            question   : question,
            anchor     : '100%',
            xtype      : 'textfield'
        }, question.get('config'));

        fields.push(field);
    });

    form.removeAll();
    form.add({
        xtype : 'fieldset',
        title : record.get('name'),
        items : fields,
        width : '100%'
    });

    form.add({
        xtype  : 'button',
```

❷ 建立组开关意识

❸ 监听问题字段变化

❹ 处理组选择

❺ 基于问题数据准备表单字段

❻ 为问题字段的容器添加字段集

❼ 添加开关按钮便捷组

```
            text   : isLastGroup ? 'Save' : 'Next',
            itemId : isLastGroup ? 'surveyFinish' : 'groupNext',
            width  : 200
        });
    },

    showNextGroup : function () {                    ⑧ 设置开关自动化
        var grid = this.getGroups(),
            store = grid.getStore(),
            selModel = grid.getSelectionModel(),
            selected = selModel.getLastSelected(),
            curIndex = store.indexOf(selected),
            next = store.getAt(curIndex + 1);

        if (next) {
            selModel.select([next]);
        }
    },
    finishSurvey : function () {                     ⑨ 结束调查,重置调查视图
        var groups = this.getGroups();
        this.getForm().removeAll();
        groups.getSelectionModel().deselectAll();
        groups.hide();
        groups.up().down('surveylist').getSelectionModel().deselectAll();
    },
                                                     ⑩ 保存问题值
    saveItem : function (field) {
        var question = field.question;

        if (!question) {
            field = field.up('[question]');
            question = field.question;
        }

        if (question) {
            question.set('answer', field.getValue());
        }
    }
});
```

如代码清单14-16所示，Questions控制器做了很多工作。它从一个视图依赖开始❶，并且你在Surveys控制器中已经定义了这个视图。重复定义依赖不是坏习惯，请放心，加载器不会把它加载两次的。重新定义依赖关系让你能够创建可重用的代码，同时确保其他开发人员知道控制器在处理哪些类。

this.control板块罗列了4个监听器。它通过监听GroupList项的选择事件来完成和Surveys控制器的交互❷。Surveys控制器强制选择前面勾选的调查的第一个组。就是在这个时候，Questions控制器接管并处理组的选择❹。它会先遍历选择的组中所有可用的问题，然后创建Ext.form.Fields的一个数组。然后，它删除QuestionsForm中嵌套的组件。如果可以的话，给Ext.form.FieldSet留点空间，它是装载问题的容器❻。方便的一点是，字段集的名称和选择的

组的名字一样。在此时，QuestionsForm的标题显示为调查名称，字段集相对应地显示为组名称。这对于让用户知道他们在填充什么很有用。字段集的item属性指向步骤❺中聚集的一组问题。

❼中在字段集下添加了一个很有用的按钮。如果活跃的调查中有更多要访问的组，按钮会切换到下一个上❽。否则，它就会把状态重新恢复到仅展示调查列表的默认视图并结束调查❾。

问题是数据驱动的，这意味着它们是从服务器获取的。大多数情况下，它们和表单构建器应用关联。你希望数据被不断保存，而不需要用户点击一个专门的按钮，这意味着监听存在于所有字段上的一个通用事件：change事件❸。它的处理程序简单地访问选中字段的getValue方法，并把它的值保存到问题的模型实例上❿。但是这里有一个“疑难杂症”。复杂的字段，比如radiogroup和checkboxgroup，对这一事件的回应不同。它们内嵌的子元素会触发变更事件，而组容器读取值和赋值。在这种情况，你需要进一步看看如何构建一个事件触发的字段并获取getValue方法。

瞧，这个应用程序就绪，我们可以对其进行测试了。准备好了吗？让我们来启动它。图14-10展示了一次调查的客户端视图。他们选择了一个调查，本次调查会自动地选择第一个可用的组，继而在页面的中间部分加载应用的适用问题。他们的改变会被自动保存，并加载前面问题的答案。你可以通过点击Next（下一步）按钮或者从列表中选择另一个组，无缝地切换到另一个组。

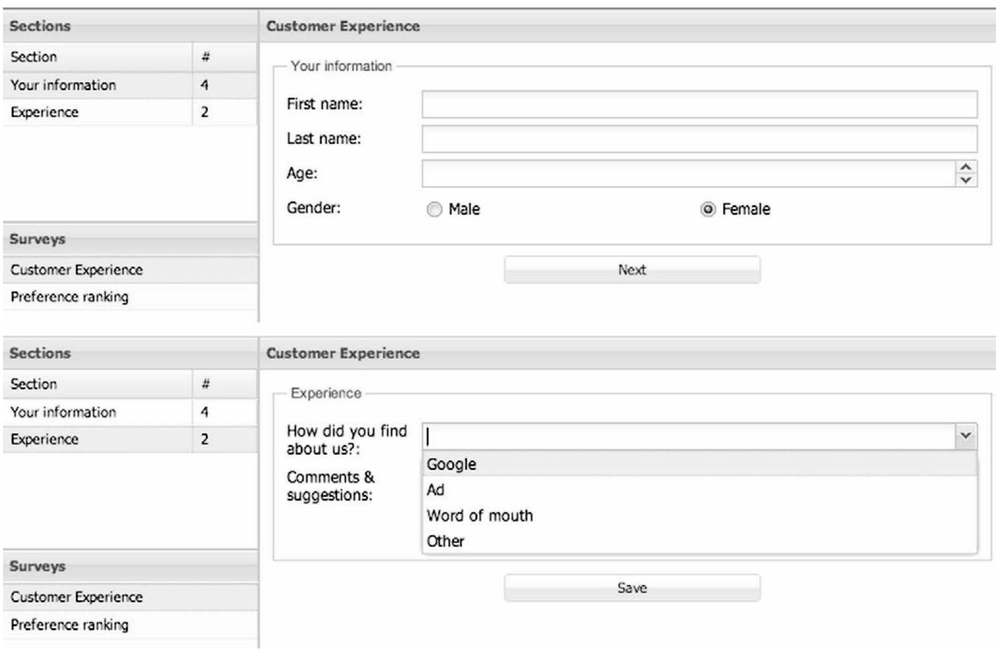


图14-10　最终调查应用的演示

恭喜你，目标应用程序完成了！可以看到，它是一个进一步开发Ext JS应用程序的很棒的测试平台。如果想了解更多，这里罗列了一些提升调查程序的理念：

- 在服务器上搭建CRUD的工作流；
- 用Ext JS图表来添加报表；
- 用表单构建器创建调查管理接口；
- 应用定制化的样式。

因为我们决定要把这个作为该应用程序的稳定版本，所以要把它打包用于生产。为此，你需要让Sencha Cmd的构建过程工作起来。

14.4　打包

打包是为最佳交付准备一个Web应用程序的一道工序。Sencha Cmd使打包变得快捷轻松。除了其他一些不错的功能，Sench Cmd还能够提供如下帮助：

- 降低包的大小，使网络传输更快；
- 提升执行速度；
- 降低内存占用。

它通过如下手段来达到这些目标：

- 去除不必要的部分，比如类、组件，甚至方法调用；
- 级联JavaScript和CSS代码；
- 削减JavaScript和CSS代码。

通过Sencha Cmd创建应用程序的初始步骤将被证明效果明显，因为现在是时候打包了。

大多数情况下，应用程序都应该为被打包做好准备。但是你必须引入一个额外的文件——data.json，所以附加的配置文件是有必要的。当然，你应该迁移到RESTful（或者类似的客户端-服务器端）的环境中，这里可以跳过这一步。

为了引入构建过程中需要的文件，你必须对app.json做一点修改。这一JSON对象已经有一点预先生成的内容了，你可以在它的基础上附加一个叫作`resources`的数组属性。`resources`属性告诉Sencha Cmd从相应的位置复制应用程序构建版本的文件和文件夹。下面是添加了data.json后文件的样子：

```
{
    "name": "Survey",

    "requires": [
    ],

    "resources": [
        "data.json"
    ],

    "id": "c17ccff8-b8c9-4fb1-80f5-8f818603a5e5"
}
```

你已经准备就绪可以构建应用程序了。使用Sencha Cmd，Ext JS应用程序有以下两种构建

类型。

- **测试**　以最少的优化级联JS和CSS文件。
- **生产**　全面优化并压缩的JavaScript和CSS文件。

由于两种构建类型的构建过程都是相同的，我们直接跳到生产版本的构建上。打开终端，然后导航到调查应用程序的根目录下。现在执行如下命令：

```
sencha app build production
```

请高枕无忧地看着程序启动，它会检查JavaScript代码的错误、编译Sass主题、级联并压缩代码，并且还会做其他很多事情来优化应用程序，以实现最佳的性能和用户体验。

如图14-11所验证的，Sencha Cmd在构建目录下创建了一个新的目录结构，构建目录包含了调查应用的生产版本。让我们看看构建后的版本和原始开发的源代码有什么区别。

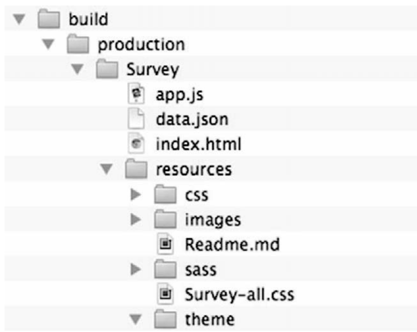


图14-11　构建完成的应用程序文件

从运行速度上来说，大体的检测结果如下。

- 未构建（开发）版本的执行时间：1764.207 ms。
- 完全构建（生产）版本的执行时间：472.525 ms。

令人印象深刻的是生产版本的速度超过开发版本两倍。实际上是快3.7倍，假如你是在网络状况较差的情况下从远程地址下载应用，这个数字还会更大。总大小怎么样呢？

在Chrome Inspector（或者其他任何网络元素审查工具）上使用Network（网络）选项卡，你可以看到两个版本的不同之处，如下所述。

- 未构建（开发）版本的传输：3.8 M每274次请求。
- 完全构建（生产）版本的传输：1.2 M每4次请求。

生产类型的构建毫无争议是为终端用户提供服务的更好方式。它执行速度更快，并且下载速度更快，使用更少的昂贵请求。

在一个常规的应用开发周期中，这是质量验收测试之后的最后一个测试了。这是庆贺的最佳时间！享受本地和远程的测试，并体验Sencha Cmd构建过程给你从头开始构建应用程序所带来的益处吧。

14.5　小结

在这紧凑的一章中，你探索了构建应用程序的主要步骤。不管是小型还是大型应用，开发流程都是一样的。11-SAW流程对于规划开发步骤非常有用。请时刻谨记代码约定对于产品生命周期的重要性。请使用本章描述的Ext JS规范，并用于开发JavaScript和CSS代码。

我们通过本书讲解了大量内容。你了解了框架的内部机制、部件、类系统、MVC模式、Sencha Cmd构建过程等。该框架将继续存在，并且Sencha会继续开发新的激动人心的功能来丰富它。或许，我们最希望你掌握钻研Ext JS源代码的能力，以便能够持续解决无穷尽的问题。